O'REILLY®

Python 机器学习手册

从数据预处理到深度学习

Machine Learning with Python Cookbook

[美]Chris Albon 著
韩慧昌 林然 徐江 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

内 容 简 介

本书采用基于任务的方式来介绍如何在机器学习中使用Python。书中有近200个独立的解决方案，针对的都是数据科学家或机器学习工程师在构建模型时可能遇到的常见任务，涵盖从简单的矩阵和向量运算到特征工程以及神经网络的构建。所有方案都提供了相关代码，读者可以复制并粘贴这些代码，用在自己的程序中。

本书不是机器学习的入门书，适合熟悉机器学习理论和概念的读者阅读。你可以将本书作为案头参考书，在机器学习的日常开发中遇到问题时，随时借鉴书中代码，快速解决问题。

©2018 by Chris Albon
Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2019. Authorized translation of the English edition, 2018 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.
All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有出版权由O'Reilly Media, Inc. 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号　图字：01-2018-5011

图书在版编目（CIP）数据

Python机器学习手册：从数据预处理到深度学习 /（美）克里斯·阿尔本（Chris Albon）著；韩慧昌，林然，徐江译. —北京：电子工业出版社，2019.7
书名原文：Machine Learning with Python Cookbook
ISBN 978-7-121-36962-9

Ⅰ. ①P… Ⅱ. ①克… ②韩… ③林… ④徐… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字（2019）第124840号

责任编辑：许　艳
封面设计：Karen Montgomery　张　健
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编：100036
开　　本：787×980　1/16　印张：22.75　字数：498千字
版　　次：2019年7月第1版
印　　次：2019年7月第1次印刷
定　　价：89.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

序

在过去的几年中，机器学习已经渗透到企业、非营利组织和政府的日常运作中。随着机器学习热度的增加，在对机器学习从业者的指导方面，涌现出一批高质量的文献。这些文献培养了整整一代的数据科学家和机器学习工程师。这些文献提供了学习资源，为人们讲解机器学习是什么及其工作原理。尽管这种方法富有成效，但却遗漏了一部分内容：机器学习日常开发中的细节。这就是笔者写本书的动机——本书不是写给学生读者学习机器学习理论的大部头，而是写给专业人士的“扳手型”工具书。我希望你把它放在书桌上，把你感兴趣的某些页折起来，在日常开发中需要解决实际问题时就拿过来翻一翻。

更具体地说，本书采用基于任务的方式来介绍机器学习，有近 200 个独立的解决方案（你可以复制并粘贴这些代码，它们都是可以正常运行的），针对的都是数据科学家或机器学习工程师在构建模型时可能遇到的常见任务。

本书的最终目标是成为人们在构建真实的机器学习系统时的参考书。例如，假设你有一个 JSON 文件，其中包含 1000 个数据分类特征和数值型特征，并且目标向量的分类不均衡，你想得到一个可解释的模型，那么使用本书提供的解决方案可以帮助你解决如下问题：

- 加载 JSON 文件（2.5 节）
- 对特征进行标准化（4.2 节）
- 对特征字典编码（5.3 节）
- 填充缺失的分类值（5.4 节）
- 使用主成分进行特征降维（9.1 节）
- 使用随机搜索选择最佳模型（12.2 节）
- 训练随机森林分类器（14.4 节）
- 选择随机森林中的重要特征（14.7 节）

本书的目标是让你：

1．复制 / 粘贴代码，并确信它们能很好地运行在玩具数据集（toy dataset）[1] 上。

2．阅读每一节后面的“讨论”以增进对代码背后理论的理解，并了解哪些参数需要重点考虑。

3．对书中的代码进行各种组合与修改，以构建实际的应用。

哪些人适合阅读本书

本书不是机器学习的入门书。如果你对机器学习的基本概念还不太了解，或者从未花时间学习过机器学习，请不要购买本书。本书适合机器学习的实践者阅读，他们熟悉机器学习的理论和概念，可以参考书中的代码快速解决在日常开发中遇到的问题。

另外，本书假定读者熟悉 Python 编程语言和包管理。

哪些人不适合阅读本书

如前所述，本书不是机器学习的入门书，因此它不应该是你的第一本机器学习书。如果你还不熟悉交叉验证、随机森林和梯度下降等概念，那么建议你先阅读一本入门类机器学习图书，再阅读本书来学习机器学习的实用解决方案。

书中用到的术语

机器学习会用到很多领域的技术，包括计算机科学、统计学和数学，因此在关于机器学习的讨论中会使用各种各样的术语：

观察值[2]（*observation*）

我们观察到的单个单位——例如，一个人、一次销售或一条记录。

学习算法（*learning algorithm*）

用来学习模型的最佳参数的算法——例如，线性回归、朴素贝叶斯或决策树。

模型（*model*）

学习算法的输出。学习算法训练出的模型可以用来做预测。

1　译者注：可以理解为小型数据集，比真实世界的大数据集小得多、干净得多。
2　译者注：也可理解为样本，本书中某些地方将 observation 译作“样本”。

参数（*parameter*）

一个模型在训练过程中学习到的权重或系数。

超参数（*hyperparameter*）

一个学习算法在训练前需要设置的一组参数。

性能（*performance*）

用来评估模型的指标。

损失（*loss*）

一个需要在训练中最小化或最大化的指标。

训练（*train*）

使用类似梯度下降之类的数学方法将一个学习算法应用到数据上。

拟合（*fit*）

使用分析方法将一个机器学习算法应用到数据上。

数据（*data*）

一组观察值。

致谢

没有一些朋友和陌生人的帮助，这本书是不可能完成的。很难列出提供过帮助的所有人的名字，但我想至少提一下这些人：Angela Bassa、Teresa Borcuch、Justin Bozonier、Andre deBruin、Numa Dhamani、Dan Friedman、Joel Grus、Sarah Guido、Bill Kambouroglou、Mat Kelcey、Lizzie Kumar、Hilary Parker、Niti Paudyal、Sebastian Raschka 和 Shreya Shankar。

我欠他们所有人一瓶啤酒，或者五瓶。

目录

第 1 章　向量、矩阵和数组 1

1.0　简介 1
1.1　创建一个向量 1
1.2　创建一个矩阵 2
1.3　创建一个稀疏矩阵 3
1.4　选择元素 5
1.5　展示一个矩阵的属性 6
1.6　对多个元素同时应用某个操作 7
1.7　找到最大值和最小值 8
1.8　计算平均值、方差和标准差 9
1.9　矩阵变形 10
1.10　转置向量或矩阵 11
1.11　展开一个矩阵 12
1.12　计算矩阵的秩 13
1.13　计算行列式 14
1.14　获取矩阵的对角线元素 14
1.15　计算矩阵的迹 15
1.16　计算特征值和特征向量 16
1.17　计算点积 17
1.18　矩阵的相加或相减 18
1.19　矩阵的乘法 19

1.20 计算矩阵的逆 20
1.21 生成随机数 21

第 2 章 加载数据 23

2.0 简介 23
2.1 加载样本数据集 23
2.2 创建仿真数据集 25
2.3 加载 CSV 文件 28
2.4 加载 Excel 文件 29
2.5 加载 JSON 文件 29
2.6 查询 SQL 数据库 31

第 3 章 数据整理 33

3.0 简介 33
3.1 创建一个数据帧 34
3.2 描述数据 35
3.3 浏览数据帧 37
3.4 根据条件语句来选择行 39
3.5 替换值 40
3.6 重命名列 41
3.7 计算最小值、最大值、总和、平均值与计数值 43
3.8 查找唯一值 44
3.9 处理缺失值 45
3.10 删除一列 47
3.11 删除一行 48
3.12 删除重复行 49
3.13 根据值对行分组 51
3.14 按时间段对行分组 52
3.15 遍历一个列的数据 54
3.16 对一列的所有元素应用某个函数 55
3.17 对所有分组应用一个函数 56
3.18 连接多个数据帧 57
3.19 合并两个数据帧 59

第 4 章　处理数值型数据 63
4.0　简介 63
4.1　特征的缩放 63
4.2　特征的标准化 65
4.3　归一化观察值 66
4.4　生成多项式和交互特征 69
4.5　转换特征 70
4.6　识别异常值 71
4.7　处理异常值 73
4.8　将特征离散化 75
4.9　使用聚类的方式将观察值分组 77
4.10　删除带有缺失值的观察值 79
4.11　填充缺失值 81
第 5 章　处理分类数据 83
5.0　简介 83
5.1　对 nominal 型分类特征编码 84
5.2　对 ordinal 分类特征编码 86
5.3　对特征字典编码 88
5.4　填充缺失的分类值 91
5.5　处理不均衡分类 93
第 6 章　处理文本 97
6.0　简介 97
6.1　清洗文本 97
6.2　解析并清洗 HTML 99
6.3　移除标点 100
6.4　文本分词 101
6.5　删除停止词（stop word） 102
6.6　提取词干 103
6.7　标注词性 104
6.8　将文本编码成词袋（Bag of Words） 107
6.9　按单词的重要性加权 109

第 7 章　处理日期和时间 113
7.0　简介 113
7.1　把字符串转换成日期 113
7.2　处理时区 115
7.3　选择日期和时间 116
7.4　将日期数据切分成多个特征 117
7.5　计算两个日期之间的时间差 118
7.6　对一周内的各天进行编码 119
7.7　创建一个滞后的特征 120
7.8　使用滚动时间窗口 121
7.9　处理时间序列中的缺失值 123
第 8 章　图像处理 127
8.0　简介 127
8.1　加载图像 128
8.2　保存图像 130
8.3　调整图像大小 131
8.4　裁剪图像 132
8.5　平滑处理图像 133
8.6　图像锐化 136
8.7　提升对比度 138
8.8　颜色分离 140
8.9　图像二值化 142
8.10　移除背景 144
8.11　边缘检测 148
8.12　角点检测 150
8.13　为机器学习创建特征 153
8.14　将颜色平均值编码成特征 156
8.15　将色彩直方图编码成特征 157
第 9 章　利用特征提取进行特征降维 161
9.0　简介 161
9.1　使用主成分进行特征降维 161

9.2 对线性不可分数据进行特征降维 164
9.3 通过最大化类间可分性进行特征降维 166
9.4 使用矩阵分解法进行特征降维 169
9.5 对稀疏数据进行特征降维 170

第 10 章 使用特征选择进行降维 173

10.0 简介 173
10.1 数值型特征方差的阈值化 173
10.2 二值特征的方差阈值化 175
10.3 处理高度相关性的特征 176
10.4 删除与分类任务不相关的特征 178
10.5 递归式特征消除 180

第 11 章 模型评估 183

11.0 简介 183
11.1 交叉验证模型 183
11.2 创建一个基准回归模型 187
11.3 创建一个基准分类模型 188
11.4 评估二元分类器 190
11.5 评估二元分类器的阈值 193
11.6 评估多元分类器 197
11.7 分类器性能的可视化 198
11.8 评估回归模型 201
11.9 评估聚类模型 203
11.10 创建自定义评估指标 204
11.11 可视化训练集规模的影响 206
11.12 生成对评估指标的报告 208
11.13 可视化超参数值的效果 209

第 12 章 模型选择 213

12.0 简介 213
12.1 使用穷举搜索选择最佳模型 213
12.2 使用随机搜索选择最佳模型 216
12.3 从多种学习算法中选择最佳模型 218

12.4 将数据预处理加入模型选择过程 220
12.5 用并行化加速模型选择 221
12.6 使用针对特定算法的方法加速模型选择 223
12.7 模型选择后的性能评估 224

第 13 章 线性回归 227

13.0 简介 227
13.1 拟合一条直线 227
13.2 处理特征之间的影响 229
13.3 拟合非线性关系 231
13.4 通过正则化减少方差 233
13.5 使用套索回归减少特征 235

第 14 章 树和森林 237

14.0 简介 237
14.1 训练决策树分类器 237
14.2 训练决策树回归模型 239
14.3 可视化决策树模型 240
14.4 训练随机森林分类器 243
14.5 训练随机森林回归模型 244
14.6 识别随机森林中的重要特征 245
14.7 选择随机森林中的重要特征 248
14.8 处理不均衡的分类 249
14.9 控制决策树的规模 250
14.10 通过 boosting 提高性能 252
14.11 使用袋外误差（Out-of-Bag Error）评估随机森林模型 253

第 15 章 KNN 255

15.0 简介 255
15.1 找到一个观察值的最近邻 255
15.2 创建一个 KNN 分类器 258
15.3 确定最佳的邻域点集的大小 260
15.4 创建一个基于半径的最近邻分类器 261

第 16 章　逻辑回归 263
16.0　简介 263
16.1　训练二元分类器 263
16.2　训练多元分类器 265
16.3　通过正则化来减小方差 266
16.4　在超大数据集上训练分类器 267
16.5　处理不均衡的分类 269
第 17 章　支持向量机 271
17.0　简介 271
17.1　训练一个线性分类器 271
17.2　使用核函数处理线性不可分的数据 274
17.3　计算预测分类的概率 278
17.4　识别支持向量 279
17.5　处理不均衡的分类 281
第 18 章　朴素贝叶斯 283
18.0　简介 283
18.1　为连续的数据训练分类器 284
18.2　为离散数据和计数数据训练分类器 286
18.3　为具有二元特征的数据训练朴素贝叶斯分类器 287
18.4　校准预测概率 288
第 19 章　聚类 291
19.0　简介 291
19.1　使用 K-Means 聚类算法 291
19.2　加速 K-Means 聚类 294
19.3　使用 Meanshift 聚类算法 295
19.4　使用 DBSCAN 聚类算法 296
19.5　使用层次合并聚类算法 298
第 20 章　神经网络 301
20.0　简介 301
20.1　为神经网络预处理数据 302

20.2 设计一个神经网络 304
20.3 训练一个二元分类器 307
20.4 训练一个多元分类器 309
20.5 训练一个回归模型 311
20.6 做预测 313
20.7 可视化训练历史 315
20.8 通过权重调节减少过拟合 318
20.9 通过提前结束减少过拟合 320
20.10 通过 Dropout 减少过拟合 322
20.11 保存模型训练过程 324
20.12 使用 k 折交叉验证评估神经网络 326
20.13 调校神经网络 328
20.14 可视化神经网络 331
20.15 图像分类 333
20.16 通过图像增强来改善卷积神经网络的性能 337
20.17 文本分类 339

第 21 章 保存和加载训练后的模型 343

21.0 简介 343
21.1 保存和加载 scikit-learn 模型 343
21.2 保存和加载 Keras 模型 345

第 1 章

向量、矩阵和数组

1.0 简介

NumPy 是 Python 机器学习技术栈的基础。NumPy 能对机器学习中常用的数据结构——向量（vector）、矩阵（matrice）、张量（tensor）——进行高效的操作。NumPy 并不是本书的重点，不过它在后面的章节中也会频繁出现。本章将介绍在进行机器学习的过程中可能经常遇到的 NumPy 操作。

1.1 创建一个向量

问题描述

创建一个向量。

解决方案

使用 NumPy 创建一个一维数组：

```
# 加载库
import numPy as np

# 创建一个行向量
vector_row = np.array([1, 2, 3])

# 创建一个列向量
vector_column = np.array([[1],
                          [2],
                          [3]])
```

讨论

NumPy 的主要数据结构是多维数组。要创建一个向量，只需简单地创建一个一维数组即可。和向量相似，对这些数组，我们能水平地（也就是行）或垂直地（也就是列）表示。

延伸阅读

- Vectors（Math Is Fun 网站，*http://bit.ly/2FB5q1v*）
- 欧几里得向量（维基百科，*http://bit.ly/2FtnRoL*）

1.2 创建一个矩阵

问题描述

创建一个矩阵。

解决方案

使用 NumPy 创建一个二维数组：

```
# 加载库
import numPy as np

# 创建一个矩阵
matrix = np.array([[1, 2],
                   [1, 2],
                   [1, 2]])
```

讨论

可以通过创建一个 NumPy 二维数组来创建一个矩阵。在上述解决方案中，我们的矩阵有三行两列（一列的元素都是 1，另一列都是 2）。

实际上，NumPy 提供了专用的数据结构来表示矩阵：

```
matrix_object = np.mat([[1, 2],
                        [1, 2],
                        [1, 2]])

matrix([[1, 2],
        [1, 2],
        [1, 2]])
```

但是，我们不推荐使用矩阵数据结构。有两个原因，一是实际上数组才是 NumPy 的标准数据结构，二是绝大多数 NumPy 操作返回的是数组而不是矩阵对象。

延伸阅读

- 矩阵（维基百科，*http://bit.ly/2Ftnevp*）
- 矩阵（Wolfram MathWorld，*http://bit.ly/2Fut7IJ*）

1.3 创建一个稀疏矩阵

问题描述

高效地表示只有零星非零值的数据。

解决方案

创建一个稀疏矩阵：

```
# 加载库
import numpy as np
from scipy import sparse

# 创建一个矩阵
matrix = np.array([[0, 0],
                   [0, 1],
                   [3, 0]])

# 创建一个压缩的稀疏行（Compressed Sparse Row，CSR）矩阵
matrix_sparse = sparse.csr_matrix(matrix)
```

讨论

在机器学习中，数据集十分庞大且其中大部分元素都是零的情况很常见。举个例子，想象一个矩阵，列元素是 Netflix 上的所有电影，行元素是 Netflix 的所有用户，矩阵中元素的值代表对于某部电影每个用户看过几次。这个矩阵将会拥有成千上万列以及数百万行！但由于大多数用户只看过一部分电影，所以绝大多数元素的值是零。

稀疏矩阵只保存非零元素并假设剩余元素的值都是零，这样能节省大量的计算成本。在上述解决方案中，我们创建了带有两个非零值的 NumPy 数组，然后将其转换成一个稀疏矩阵。如果查看这个稀疏矩阵，就能看到只有其中的非零值被保存下来了：

```
# 查看稀疏矩阵
print(matrix_sparse)

    (1, 1)    1
    (2, 0)    3
```

稀疏矩阵的类型有很多。在*压缩的稀疏行*（CSR）矩阵中，(1,1) 和 (2, 0) 使用下标（下标的编号从 0 开始递增）分别表示非零值 1 和 3。例如，元素 1 就位于矩阵的第二行和第二列。如果创建一个更大的矩阵，而这个矩阵有更多的非零元素，那么与普通矩阵相比，稀疏矩阵的优势就体现出来了：

```
# 创建一个更大的矩阵
matrix_large = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                         [3, 0, 0, 0, 0, 0, 0, 0, 0, 0]])

# 创建 CSR 矩阵
matrix_large_sparse = sparse.csr_matrix(matrix_large)

# 查看原先的稀疏矩阵
print(matrix_sparse)

  (1, 1)    1
  (2, 0)    3

# 查看更大的稀疏矩阵
print(matrix_large_sparse)

  (1, 1)    1
  (2, 0)    3
```

如上所示，尽管实际上这个更大的矩阵中增加了很多非零元素，但是它的表示和原来那个稀疏矩阵是一模一样的。也就是说，增加的零元素没有改变稀疏矩阵的大小。

前面说过，稀疏矩阵的类型有很多，比如，压缩的稀疏列、表中表以及键值对词典。对稀疏矩阵的类型及含义的介绍超出了本书的范围，但是值得注意的是，不存在“完美”的稀疏矩阵类型，它们之间的差异都是有意义的，而且知道在什么场景下选择哪种类型的稀疏矩阵是很有必要的。

延伸阅读

- SciPy 文档：稀疏矩阵（*http://bit.ly/2HReBZR*）
- 《存储稀疏矩阵的 101 种方式》（*http://bit.ly/2HS43c1*）

1.4 选择元素

问题描述

在向量或矩阵中选择一个或多个元素。

解决方案

使用 NumPy 的数组来解决这个问题就很简单：

```
# 加载库
import numpy as np

# 创建一个行向量
vector = np.array([1, 2, 3, 4, 5, 6])

# 创建矩阵
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# 选择向量的第三个元素
vector[2]

3

# 选择第二行第二列
matrix[1,1]

5
```

讨论

如同 Python 中的大多数事物一样，NumPy 数组的索引编号是从 0 开始的，这意味着第一个元素的下标是 0 而不是 1。此外，NumPy 还提供了很多方式来选取元素或数组中的一组元素（即索引和切片）：

```
# 选取一个向量的所有元素
vector[:]

array([1, 2, 3, 4, 5, 6])

# 选取从 0 开始一直到第 3 个（包括第 3 个）元素
vector[:3]
```

```
array([1, 2, 3])

# 选取第 3 个元素之后的所有元素
vector[3:]

array([4, 5, 6])

# 选取最后一个元素
vector[-1]

6

# 选取矩阵的第 1 行和第 2 行以及所有列
matrix[:2,:]

array([[1, 2, 3],
       [4, 5, 6]])

# 选取所有行以及第二列
matrix[:,1:2]

array([[2],
       [5],
       [8]])
```

1.5 展示一个矩阵的属性

问题描述

展示一个矩阵的形状、大小和维数。

解决方案

使用 shape、size 和 ndim 函数：

```
# 加载库
import numpy as np

# 创建矩阵
matrix = np.array([[1, 2, 3, 4],
                   [5, 6, 7, 8],
                   [9, 10, 11, 12]])
```

```
# 查看行数和列数
matrix.shape
(3, 4)

# 查看元素的数量（行数 * 列数）
matrix.size

12

# 查看维数
matrix.ndim

2
```

讨论

虽然这些操作看起来很简单（确实也是这样），但是在做下一步计算之前或在某个操作之后，简单检查一遍数组的形状和大小是很有价值的。

1.6 对多个元素同时应用某个操作

问题描述

对一个数组中的多个元素同时应用某个函数。

解决方案

使用 NumPy 的 vecotrize：

```
# 加载库
import numpy as np

# 创建一个矩阵
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# 创建一个函数，返回输入值加上 100 以后的值
add_100 = lambda i: i + 100

# 创建向量化函数
vectorized_add_100 = np.vectorize(add_100)
```

```
# 对矩阵的所有元素应用这个函数
vectorized_add_100(matrix)

array([[101, 102, 103],
       [104, 105, 106],
       [107, 108, 109]])
```

讨论

NumPy 的 vectorize 类将一个函数转换成另一个函数，这个函数能把某个操作应用在数组的全部元素或一个切片上。值得注意的是，vectorize 本质上是在对所有元素循环执行某个操作，所以并不会提升性能。此外，使用 NumPy 的数组，我们可以对两个维度不同的数组执行操作（这是一种叫作广播的方法）。举个例子，对于上述问题使用广播（broadcasting）后，我们的解决方案更简单：

```
# 将所有的元素加上 100
matrix + 100

array([[101, 102, 103],
       [104, 105, 106],
       [107, 108, 109]])
```

1.7 找到最大值和最小值

问题描述

计算一个数组的最大值或最小值。

解决方案

使用 NumPy 的 max 和 min 方法：

```
# 加载库
import numpy as np

# 创建一个矩阵
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# 返回最大的元素
np.max(matrix)
```

```
9

# 返回最小的元素
np.min(matrix)

1
```

讨论

求一个数组或一个数组的子集中元素的最大值和最小值是很常见的需求，使用 `max` 和 `min` 方法很容易实现。而使用 `axis` 参数则可以对一个特定的坐标轴应用此操作：

```
# 找到每一列最大的元素
np.max(matrix, axis=0)

array([7, 8, 9])

# 找到每一行最大的元素
np.max(matrix, axis=1)

array([3, 6, 9])
```

1.8 计算平均值、方差和标准差

问题描述

计算数组的一些描述性统计值。

解决方案

使用 NumPy 的 `mean`、`var` 和 `std`：

```
# 加载库
import numpy as np

# 创建一个矩阵
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# 返回平均值
np.mean(matrix)

5.0
```

```
# 返回方差
np.var(matrix)

6.666666666666667

# 返回标准差
np.std(matrix)

2.5819888974716112
```

讨论

就如同使用 max 和 min 一样，我们很容易就可以计算出整个矩阵或其中一个坐标轴的描述性统计值：

```
# 求每一列的平均值
np.mean(matrix, axis=0)

array([ 4.,  5.,  6.])
```

1.9 矩阵变形

问题描述

在不改变元素值的前提下，改变一个数组的形状（行数和列数）。

解决方案

使用 NumPy 的 reshape：

```
# 加载库
import numpy as np

# 创建一个 4 x 3 的矩阵
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9],
                   [10, 11, 12]])

# 将该矩阵变形为 2 x 6 的矩阵
matrix.reshape(2, 6)

array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

讨论

reshape可以重构一个数组，维持该数组原来的数据不变，只改变行数和列数。唯一的要求就是原矩阵和新矩阵包含的元素数量必须相同（也就是大小相同）。可以通过size来查看矩阵的大小：

```
matrix.size
```

```
12
```

reshape能传入一个非常有用的参数值-1，表示可以“根据需要填充元素”，所以reshape(-1, 1)意味着矩阵的行数是1，而列数则根据需要填充：

```
matrix.reshape(1, -1)
```

```
array([[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]])
```

最后，如果只提供一个整数作为参数，那么reshape会返回一个长度为该整数值的一维数组：

```
matrix.reshape(12)
```

```
array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

1.10 转置向量或矩阵

问题描述

转置一个向量或矩阵。

解决方案

使用T方法：

```
# 加载库
import numpy as np

# 创建一个矩阵
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# 转置一个矩阵
matrix.T
```

```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

讨论

转置在线性代数中是很常见的操作，它将每个元素的行、列下标互换。在线性代数范围外有一点常被忽视：严格来说一个向量是不能被转置的，因为它只是值的集合。

```
# 将向量转置
np.array([1, 2, 3, 4, 5, 6]).T

array([1, 2, 3, 4, 5, 6])
```

转置一个向量通常是指将行向量转换为列向量（注意第 2 对括号），或反向转换：

```
# 转置一个行向量
np.array([[1, 2, 3, 4, 5, 6]]).T

array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

1.11 展开一个矩阵

问题描述

将矩阵转换成一个一维数组。

解决方案

使用 flatten：

```
# 加载库
import numpy as np

# 创建一个矩阵
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
```

```
# 将矩阵展开
matrix.flatten()

array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

讨论

flatten 是将矩阵转换成一维数组的一种简单方法。另一种方法是使用 reshape 来创建一个行向量：

```
matrix.reshape(1, -1)

array([[1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

1.12 计算矩阵的秩

问题描述

计算一个矩阵的秩。

解决方案

使用 NumPy 中的线性代数方法 matrix_rank：

```
# 加载库
import numpy as np

# 创建一个矩阵
matrix = np.array([[1, 1, 1],
                   [1, 1, 10],
                   [1, 1, 15]])

# 返回矩阵的秩
np.linalg.matrix_rank(matrix)

2
```

讨论

矩阵的秩就是由它的列或行展开的向量空间的维数。多亏有 NumPy 的 matrix_rank，计算一个矩阵的秩才能如此简单。

延伸阅读

- 《矩阵的秩》（CliffsNotes，*http://bit.ly/2HUzkMs*）

1.13 计算行列式

问题描述

计算一个矩阵的行列式。

解决方案

使用 NumPy 的线性代数方法 `det`：

```
# 加载库
import numpy as np

# 创建一个矩阵
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]])

# 返回矩阵的行列式
np.linalg.det(matrix)

0.0
```

讨论

有时候矩阵的行列式是很有用的，使用 NumPy 的 `det` 能很容易地计算出矩阵的行列式。

延伸阅读

- 《线性代数的本质》，第 5 章“行列式”（3Blue1Brown，*http:// bit.ly/2FA6ToM*）
- 行列式（Wolfram MathWorld，*http://bit.ly/2FxSUzC*）

1.14 获取矩阵的对角线元素

问题描述

获取矩阵的对角线元素。

解决方案

使用 diagonal：

```
# 加载库
import numpy as np

# 创建一个矩阵
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]])

# 返回对角线元素
matrix.diagonal()

array([1, 4, 9])
```

讨论

使用 NumPy 的 diagonal 很容易获取矩阵的对角线元素。我们还可以使用 offset 参数在主对角线的上下偏移，获取偏移后的对角线方向的元素：

```
# 返回主对角线向上偏移量为 1 的对角线元素

matrix.diagonal(offset=1)

array([2, 6])

# 返回主对角线向下偏移量为 1 的对角线元素
matrix.diagonal(offset=-1)

array([2, 8])
```

1.15 计算矩阵的迹

问题描述

计算一个矩阵的迹。

解决方案

使用 trace：

```
# 加载库
```

```
import numpy as np

# 创建一个矩阵
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]]),

# 返回矩阵的迹
matrix.trace()

14
```

讨论

矩阵的迹是其对角线元素之和，常被用在机器学习方法的底层计算中。给定一个 NumPy 的多维数组，使用 `trace` 就能计算出它的迹。我们还可以通过先返回矩阵的对角线元素再对其求和的方式来计算矩阵的迹：

```
# 返回对角线元素并求和
sum(matrix.diagonal())

14
```

延伸阅读

- 《方阵的迹》（*http://bit.ly/2FunM45*）

1.16 计算特征值和特征向量

问题描述

计算一个方阵的特征值和特征向量。

解决方案

使用 NumPy 的 `linalg.eig`：

```
# 加载库
import numpy as np

# 创建一个矩阵
matrix = np.array([[1, -1, 3],
                   [1, 1, 6],
                   [3, 8, 9]])
```

```
# 计算特征值和特征向量
eigenvalues, eigenvectors = np.linalg.eig(matrix)

# 查看特征值
eigenvalues

array([ 13.55075847,   0.74003145,  -3.29078992])

# 查看特征向量
eigenvectors

array([[-0.17622017, -0.96677403, -0.53373322],
       [-0.435951  ,  0.2053623 , -0.64324848],
       [-0.88254925,  0.15223105,  0.54896288]])
```

讨论

在机器学习库中，特征向量被广泛地使用。从直觉上来说，假设线性变换是以矩阵 $\boldsymbol{A}$ 的形式给出的，则当应用此线性变换时，特征向量只会改变大小（不改变方向）。更正式的描述如下：

$$\boldsymbol{A}\boldsymbol{v} = \lambda\boldsymbol{v}$$

$\boldsymbol{A}$ 是方阵，λ 为特征值，$\boldsymbol{v}$ 为特征向量。在 NumPy 的线性代数工具集中，`eig` 能计算出任何方阵的特征值和特征向量。

延伸阅读

- 特征向量和特征值的图形化解释（Setosa.io，*http://bit.ly/2Hb32LV*）
- 《线性代数的本质》第 10 章“特征向量和特征值”（3Blue1Brown，*http://bit.ly/2HeGppK*）

1.17 计算点积

问题描述

计算两个向量的点积。

解决方案

使用 NumPy 的 `dot`：

```
# 加载库
import numpy as np

# 创建两个向量
vector_a = np.array([1,2,3])
vector_b = np.array([4,5,6])

# 计算点积
np.dot(vector_a, vector_b)

32
```

讨论

向量 ***a*** 和 ***b*** 的点积定义如下：

$$\sum_{i=1}^{n} a_i b_i$$

a_i 是向量 ***a*** 的第 i 个元素。可以使用 NumPy 的 `dot` 来计算点积，或者在 Python 3.5 以上的版本中使用新操作符 `@`：

```
# 计算点积
vector_a @ vector_b

32
```

延伸阅读

- 向量点积和向量长度（Khan Academy，*http://bit.ly/2Fr0AUe*）
- 点积（Paul's Online Math Notes，*http://bit.ly/2HgZHLp*）

1.18 矩阵的相加或相减

问题描述

将两个矩阵相加或相减。

解决方案

使用 NumPy 的 `add` 和 `subtract`：

```
# 加载库
import numpy as np
```

```
# 创建一个矩阵
matrix_a = np.array([[1, 1, 1],
                     [1, 1, 1],
                     [1, 1, 2]])

# 创建另一个矩阵
matrix_b = np.array([[1, 3, 1],
                     [1, 3, 1],
                     [1, 3, 8]])

# 将两个矩阵相加
np.add(matrix_a, matrix_b)

array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])

# 将两个矩阵相减
np.subtract(matrix_a, matrix_b)

array([[ 0, -2,  0],
       [ 0, -2,  0],
       [ 0, -2, -6]])
```

讨论

还可以简单地使用“+”和“-”操作符来实现矩阵的相加或相减：

```
# 将两个矩阵相加
matrix_a + matrix_b

array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])
```

1.19 矩阵的乘法

问题描述

将两个矩阵相乘。

解决方案

使用 NumPy 的 `dot`：

```
# 加载库
import numpy as np

# 创建一个矩阵
matrix_a = np.array([[1, 1],
                     [1, 2]])

# 创建另一个矩阵
matrix_b = np.array([[1, 3],
                     [1, 2]])

# 将两个矩阵相乘
np.dot(matrix_a, matrix_b)

array([[2, 5],
       [3, 7]])
```

讨论

还可以在 Python 3.5 以上的版本中使用“@”操作符：

```
# 将两个矩阵相乘
matrix_a @ matrix_b

array([[2, 5],
       [3, 7]])
```

如果想将两个矩阵对应的元素相乘，则需要使用“*”操作符：

```
# 让两个矩阵对应的元素相乘
matrix_a * matrix_b

array([[1, 3],
       [1, 4]])
```

延伸阅读

- 数组操作 vs. 矩阵操作（MathWorks，*http://bit.ly/2FtpXVr*）

1.20 计算矩阵的逆

问题描述

计算一个方阵的逆。

解决方案

使用 NumPy 中的线性代数方法 inv：

```
# 加载库
import numpy as np

# 创建一个矩阵
matrix = np.array([[1, 4],
                   [2, 5]])

# 计算一个矩阵的逆
np.linalg.inv(matrix)

array([[-1.66666667,  1.33333333],
       [ 0.66666667, -0.33333333]])
```

讨论

假设方阵 A 的逆是另一个矩阵 A^{-1}，则：

$$A A^{-1} = I$$

I 是单位矩阵。如果 A^{-1} 存在，则可以使用 NumPy 的 linalg.inv 来计算。为了看到效果，可以将一个矩阵和它的逆矩阵相乘，结果会是一个单位矩阵：

```
# 将矩阵和它的逆矩阵相乘
matrix @ np.linalg.inv(matrix)

array([[ 1.,  0.],
       [ 0.,  1.]])
```

延伸阅读

- 矩阵的逆（*http://bit.ly/2Fzf0BS*）

1.21 生成随机数

问题描述

生成一些伪随机数。

解决方案

使用 NumPy 的 random：

```
# 加载库
import numpy as np

# 设置随机数种子
np.random.seed(0)

# 生成 3 个 0.0 到 1.0 之间的随机浮点数
np.random.random(3)

array([ 0.5488135 ,  0.71518937,  0.60276338])
```

讨论

NumPy 提供了生成随机数的大量方法，对此更深入的讨论超出了本节的范围。我们的解决方案中生成的是浮点数，但其实生成整数更常见：

```
# 生成 3 个 1 到 10 之间的随机整数
np.random.randint(0, 11, 3)

array([3, 7, 9])
```

或者可以从一个分布中随机抽取一些数来生成随机数：

```
# 从平均值是 0.0 且标准差是 1.0 的正态分布中抽取 3 个数
np.random.normal(0.0, 1.0, 3)

array([-1.42232584,  1.52006949, -0.29139398])

# 从平均值是 0.0 且散布程度是 1.0 的 logistic 分布中抽取 3 个数
np.random.logistic(0.0, 1.0, 3)

array([-0.98118713, -0.08939902,  1.46416405])

# 从大于或等于 1.0 并且小于 2.0 的范围中抽取 3 个数
np.random.uniform(1.0, 2.0, 3)

array([ 1.47997717,  1.3927848 ,  1.83607876])
```

最后，多次返回相同的随机数对于获取可预测、可重复的结果是很有用的。如果希望多次返回同样的随机数，可以设置伪随机数生成器的“种子”（一个整数）。如果在程序中用到了随机数，则拥有相同“种子”的程序会生成同样的结果。对“种子”的使用会贯穿本书，所以本书中使用“种子”的代码所产生的随机数，与它们在你的电脑上运行时产生的随机数是相同的。

第 2 章

加载数据

2.0 简介

只要涉及机器学习，第一个步骤都是把原始数据加载到系统中。原始数据可能是日志文件、数据集文件或者数据库。此外，从多个源获取数据也很常见。本章讲解如何从不同的源（包括 CSV 文件和 SQL 数据库）加载数据，同时也会介绍几种生成符合需求的仿真实验数据的方法。尽管在 Python 生态体系中有很多加载数据的方法，但本书会着重使用 pandas 库的一些方法来加载外部数据，并使用 scikit-learn（Python 中的一个开源的机器学习库）来生成仿真数据。

2.1 加载样本数据集

问题描述

加载已有的样本数据集。

解决方案

scikit-learn 中预置了大量的流行数据集可供使用 ：

```
# 加载 scikit-learn 的数据集
from sklearn import datasets

# 加载手写数字数据集
digits = datasets.load_digits()

# 创建特征矩阵
```

```
features = digits.data

# 创建目标向量
target = digits.target

# 查看第一个样本数据
features[0]
```

```
array([  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.,   0.,   0.,  13.,
        15.,  10.,  15.,   5.,   0.,   0.,   3.,  15.,   2.,   0.,  11.,
         8.,   0.,   0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.,   0.,
         5.,   8.,   0.,   0.,   9.,   8.,   0.,   0.,   4.,  11.,   0.,
         1.,  12.,   7.,   0.,   0.,   2.,  14.,   5.,  10.,  12.,   0.,
         0.,   0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.])
```

讨论

在研究某个机器学习算法或者方法时，大部分人都不愿意把时间花在加载、转换、清洗从现实生活中得到的数据集上。幸运的是，scikit-learn 预置了一些很容易加载的常见数据集。这些数据集被称为“玩具（toy）”数据集，因为它们比现实中的数据集要小得多、干净得多。scikit-learn 中比较流行的玩具数据集有：

`load_boston`

包含 503 个波士顿房价的观察值。这是一个用于研究回归算法的优质数据集。

`load_iris`

包含 150 个鸢尾花尺寸的观察值。这是一个用于研究分类算法的优质数据集。

`load_digits`

包含 1797 个手写数字图片的观察值。这是一个用于研究图像分类算法的优质数据集。

延伸阅读

- scikit-learn 的玩具数据集（*http://bit.ly/2HS6Dzq*）
- 手写数字数据集（*http://bit.ly/2mNSEBZ*）

2.2 创建仿真数据集

问题描述

生成一个仿真数据集。

解决方案

scikit-learn 提供了很多创建仿真数据集的方法。其中有三个方法非常有用。如果需要用一个仿真数据集来做线性回归，make_regression 是一个不错的选择：

```
# 加载库
from sklearn.datasets import make_regression

# 生成特征矩阵、目标向量以及模型的系数
features, target, coefficients = make_regression(n_samples = 100,
                                                 n_features = 3,
                                                 n_informative = 3,
                                                 n_targets = 1,
                                                 noise = 0.0,
                                                 coef = True,
                                                 random_state = 1)

# 查看特征矩阵和目标向量
print('Feature Matrix\n', features[:3])
print('Target Vector\n', target[:3])

Feature Matrix
 [[ 1.29322588 -0.61736206 -0.11044703]
 [-2.793085    0.36633201  1.93752881]
 [ 0.80186103 -0.18656977  0.0465673 ]]
Target Vector
 [-10.37865986  25.5124503   19.67705609]
```

如果需要创建一个仿真数据集来做分类，可以使用 make_classification：

```
# 加载库
from sklearn.datasets import make_classification

# 生成特征矩阵和目标向量
features, target = make_classification(n_samples = 100,
                                       n_features = 3,
                                       n_informative = 3,
                                       n_redundant = 0,
```

```
                                        n_classes = 2,
                                        weights = [.25, .75],
                                        random_state = 1)

# 查看特征矩阵和目标向量
print('Feature Matrix\n', features[:3])
print('Target Vector\n', target[:3])

Feature Matrix
 [[ 1.06354768 -1.42632219  1.02163151]
 [ 0.23156977  1.49535261  0.33251578]
 [ 0.15972951  0.83533515 -0.40869554]]
Target Vector
 [1 0 0]
```

最后，如果你想要一个适合做聚类处理的数据集，scikit-learn 提供了 `make_blobs`：

```
# 加载库
from sklearn.datasets import make_blobs

# 生成特征矩阵和目标向量
features, target = make_blobs(n_samples = 100,
                              n_features = 2,
                              centers = 3,
                              cluster_std = 0.5,
                              shuffle = True,
                              random_state = 1)

# 查看特征矩阵和目标向量
print('Feature Matrix\n', features[:3])
print('Target Vector\n', target[:3])

Feature Matrix
 [[ -1.22685609   3.25572052]
 [ -9.57463218  -4.38310652]
 [-10.71976941  -4.20558148]]
Target Vector
 [0 1 1]
```

讨论

从上述解决方案能看到，`make_regression` 返回一个浮点数的特征矩阵和一个浮点数的目标向量，而 `make_classification` 和 `make_blobs` 返回的是一个浮点数的特征矩阵和一个代表分类的整数目标矩阵。

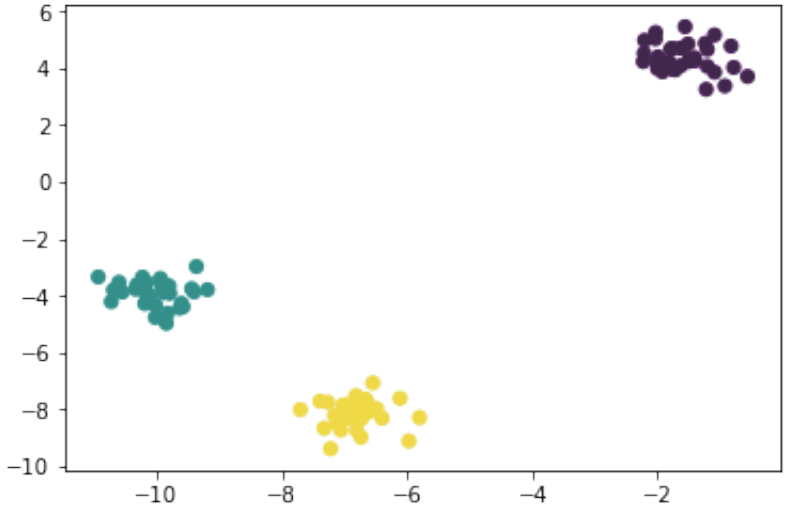
scikit-learn 的仿真数据集提供了很多选项以控制所生成数据的类型。scikit-learn 文档对所有参数都有完整的描述，但是有几个参数值得在这里特别说明。

在 make_regression 和 make_classification 中，n_informative 确定了用于生成目标向量的特征的数量。如果 n_informative 的值比总的特征数（n_features）小，则生成的数据集将包含多余的特征，这些特征可以通过特征选择技术识别出来。

另外，make_classification 包含了一个 weights 参数，可以利用它生成不均衡的仿真数据集。举个例子，如果我们设置 weights = [.25, .75]，那么生成的数据集中，25% 的观察值属于第一个分类，75% 的观察值属于第二个分类。

对于 make_blobs 来说，centers 参数决定了要生成多少个聚类。使用 matplotlib 可视化库，能将 make_blobs 生成的聚类可视化地显示出来，如下图所示。

```
# 加载库
import matplotlib.pyplot as plt

# 查看散点图
plt.scatter(features[:,0], features[:,1], c=target)
plt.show()
```



延伸阅读

- make_regression 的文档（*http://bit.ly/2FtIBwo*）
- make_classification 的文档（*http://bit.ly/2FtIKzW*）

- make_blobs 的文档（*http://bit.ly/2FqKMAZ*）

2.3 加载 CSV 文件

问题描述

加载以逗号为分隔符（Comma-Separated Values，CSV）的文件。

解决方案

使用 pandas 库的 read_csv 来加载一个本地或远端的 CSV 文件：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/simulated_data'

# 加载数据集
dataframe = pd.read_csv(url)

# 查看前两行数据
dataframe.head(2)
```

	integer	datetime	category
0	5	2015-01-01 00:00:00	0
1	5	2015-01-01 00:00:01	0

讨论

在加载 CSV 文件时，有两件事值得注意。第一，在加载数据前快速地浏览一下文件内容会很用，这便于你事先了解数据集的结构以及在加载文件时需要设置什么参数。第二，read_csv 的参数超过了 30 个，因此读文档会是一件很痛苦的事情。幸运的是，那些参数大部分是用来处理不同的 CSV 格式的。举个例子，读取 CSV 文件的字段时常依赖于一个假设，即值是由逗号分隔的（例如，可能有一行数据为 2,"2015-01-01 00:00:00",0），但是对于 CSV 文件来说，使用其他的字符作为分隔符也很常见，比如制表符。pandas 的 sep 参数可以设置文件的定界符。CSV 文件一般会有一个固定的格式（虽然也有例外）：文件的第一行指定列的数据头（例如，上述解决方案中的 integer、datetime、category）。header 参数可以指定是否存在数据头这一行以及它的位置。如果没有这一行，则需要设置 header=None。

2.4 加载 Excel 文件

问题描述

加载一个 Excel 数据表。

解决方案

使用 pandas 库的 read_excel 来加载一个 Excel 数据表：

```
# 加载库
import pandas as pd

# 创建一个 URL
url = 'https://tinyurl.com/simulated_excel'

# 加载数据
dataframe = pd.read_excel(url, sheetname=0, header=1)

# 查看前两行
dataframe.head(2)
```

	5	2015-01-01 00:00:00	0
0	5	2015-01-01 00:00:01	0
1	5	2015-01-01 00:00:02	0

讨论

这个解决方案和加载 CSV 文件的解决方案很类似。主要的区别在于多了一个参数 sheetname，它指定在加载 Excel 文件时要加载哪一张数据表。sheetname 可以是包含数据表名字的字符串，也可以是指向数据表所在位置的整数（从零开始编号）。如果需要加载多张数据表，可以把它们放在一个列表中一起传入。例如，sheetname=[0,1,2, "Monthly Sales"] 将返回一个值为 pandas DataFrame 类型的字典，该字典包含了第一张、第二张、第三张以及名为 Monthly Sales 的数据表。

2.5 加载 JSON 文件

问题描述

加载一个 JSON 文件来做数据预处理。

解决方案

pandas 库提供了 read_json，可以将 JSON 文件转换为 pandas 对象：

```
# 加载库
import pandas as pd

# 创建一个 URL
url = 'https://tinyurl.com/simulated_json'

# 加载数据
dataframe = pd.read_json(url, orient='columns')

# 查看前两行数据
dataframe.head(2)

category
datetime
integer
0
0
2015-01-01 00:00:00
5
1
0
2015-01-01 00:00:01
5
```

讨论

将 JSON 文件加载到 pandas 中，这和前两节处理的问题很相似。关键的区别在于 orient 参数，它告诉了 pandas JSON 文件的结构是什么样的。但是如果要找到与 JSON 结构正好对应的参数值（split、records、index、columns 和 values），可能需要花一些时间来做调研。pandas 还提供了一个有用的工具 json_normalize，它能将半结构化的 JSON 数据转换为 pandas 的 DataFrame 类型。

延伸阅读

- json_normalize 的文档（*http://bit.ly/2HQqwaa*）

2.6 查询 SQL 数据库

问题描述

使用结构化查询语言（Structured Query Language，SQL）从数据库中加载数据。

解决方案

pandas 的 `read_sql_query` 能在数据库中执行一个 SQL 查询语句并加载结果：

```
# 加载库
import pandas as pd
from sqlalchemy import create_engine

# 创建一个数据库的连接
database_connection = create_engine('sqlite:///sample.db')

# 加载数据
dataframe = pd.read_sql_query('SELECT * FROM data', database_connection)

# 查看前两行数据
dataframe.head(2)
```

	first_name	last_name	age	preTestScore	postTestScore
0	Jason	Miller	42	4	25
1	Molly	Jacobson	52	24	94

讨论

在本章中，本节介绍的方法很可能是在现实场景中用得最多的。SQL 是从数据库中提取数据的通用语言。在解决方案中，我们首先用 `create_engine` 定义了一个到 SQLite 数据库引擎的连接，然后用 pandas 的 `read_sql_query` 通过 SQL 语句查询数据库，并将结果存入一个 `DataFrame` 中。

SQL 是一门独立的语言，对它的讨论超出了本书的范围，但是它值得所有想学习机器学习的人深入了解。我们此处所用的 SQL 查询语句（`SELECT * FROM data`），查询了数据库并返回 `data` 表的所有列（`*`）。

延伸阅读

- SQLite（*https://www.sqlite.org*）
- W3Schools SQL 教程（*https://www.w3schools.com/sql*）

第 3 章

数据整理

3.0 简介

“数据整理（data wrangling）”是一个被广泛使用的词，经常用于描述将原始数据转换成整洁的、组织合理的形式以供使用的过程。对于我们而言，数据整理只是数据预处理中的一个步骤，但它是一个重要的步骤。

在“整理”数据时，最常用的数据结构是数据帧（data frame），它既直观又灵活。数据帧是呈表格状的，也就是说，就像你在数据表中看到的数据一样，数据帧是用行和列来表示数据的。下面给出一个用泰坦尼克号乘客的数据创建的数据帧：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 将数据作为数据帧加载进来
dataframe = pd.read_csv(url)

# 查看前 5 行数据
dataframe.head(5)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.00	female	1	1
1	Allison, Miss Helen Loraine	1st	2.00	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.00	male	0	0
3	Allison, Mrs Hudson JC(Bessie waldo Daniels)	1st	25.00	female	0	1
4	Allison, Master Hudson Trevor	1st	0.92	male	1	0

在这个数据帧中有三件重要的事情需要注意。

第一，在一个数据帧中，每一行都对应一个观察值（例如，一位乘客），每一列都对应一个特征（性别、年龄等）。举个例子，通过查看第一个观察值可知，Elisabeth Walton Allen 小姐住在头等舱，当年 29 岁，在灾难中活了下来。

第二，每一列有一个数据头（例如，`Name`、`PClass`、`Age`），每一行有一个索引（例如，这位幸运的 Elisabeth Walton Allen 小姐的索引号是 0）。这些数据能用于选择及操作观察值与特征。

第三，`Sex` 和 `SexCode` 这两列虽然用的是不同的表现形式，但是包含了同样的信息。在 `Sex` 列中，女性用字符串 `female` 表示，而在 `SexCode` 列中，女性用整数 `1` 表示。如果想让所有的特征都是唯一的，需要删除这两列中的一列。

在本章中，为了创建一组整洁、组织合理的观察值以供接下来的预处理之用，我们会调用 pandas 库，使用各种不同的方法来操作数据帧。

3.1 创建一个数据帧

问题描述

创建一个新的数据帧。

解决方案

在 pandas 中有很多方法可以创建一个新的数据帧对象。一个简单的方法是，使用 `DataFrame` 创建一个空数据帧并分别定义好每一列：

```
# 加载库
import pandas as pd

# 创建数据帧
dataframe = pd.DataFrame()

# 增加列
dataframe['Name'] = ['Jacky Jackson', 'Steven Stevenson']
dataframe['Age'] = [38, 25]
dataframe['Driver'] = [True, False]

# 查看数据帧
dataframe
```

	Name	Age	Driver
0	Jacky Jackson	38	True
1	Steven Stevenson	25	False

或者，创建完数据帧对象后，往其底部添加新的数据行：

```
# 创建一行
new_person = pd.Series(['Molly Mooney', 40, True], index=['Name','Age','Driver'])

# 附加一行
dataframe.append(new_person, ignore_index=True)
```

	Name	Age	Driver
0	Jacky Jackson	38	True
1	Steven Stevenson	25	False
2	Molly Mooney	40	True

讨论

虽然 pandas 提供了无数方法来创建数据帧，但是在真实场景中，创建一个空的数据帧对象然后再填充数据的情况几乎不会发生。相反，我们将通过加载自其他源（例如，一个 CSV 文件或数据库）的真实数据创建数据帧。

3.2 描述数据

问题描述

查看数据帧的一些属性。

解决方案

在加载完数据之后，我们能做的最简单的事之一就是使用 head 查看前几行数据：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)
```

```
# 查看前两行数据
dataframe.head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

也可以查看行和列的数量：

```
# 查看维数
dataframe.shape

(1313, 6)
```

或者，也可以使用 describe 来获取任何数值型列的描述性统计量：

```
# 查看描述性统计量
dataframe.describe()
```

	Age	Survived	SexCode
count	756.000000	1313.000000	1313.000000
mean	30.397989	0.342727	0.351866
std	14.259049	0.474802	0.477734
min	0.170000	0.000000	0.000000
25%	21.000000	0.000000	0.000000
50%	28.000000	0.000000	0.000000
75%	39.000000	1.000000	1.000000
max	71.000000	1.000000	1.000000

讨论

加载数据后，最好了解一下它的结构以及包含了什么类型的信息。理想的情况是直接查看全部数据。但是在现实中，很多情况下数据都会包含成千上万的行和列。这时，我们可以选取一些样本来查看一小部分数据，也可以利用样本来计算数据的描述性统计量。

在上述解决方案中，我们用到了由泰坦尼克号的乘客记录组成的玩具数据集。使用 head 可以查看数据集的前几行（默认查看前 5 行），使用 tail 可以查看最后几行；使用 shape 可以查看数据帧中包含了多少行和多少列；最后，使用 describe 能查看任何数值型的列的基本描述性统计量。

值得注意的是，描述性统计量并不一定能全面反映数据的情况。例如，pandas 将 Survived 和 SexCode 视为数值型的列，因为它们包含了很多 0 和 1。然而，在本例中，

数值型的值代表的是分类（例如，如果 Survived 等于 1，就表示乘客在此次海难中幸存）。由于这个原因，一些描述性统计量，比如 SexCode（代表乘客的性别）列的标准差，不能提供很直观的结论。

3.3 浏览数据帧

问题描述

从一个数据帧中挑出单个数据或一部分数据。

解决方案

使用 loc 或 iloc 能选择一个或多个数据，也能选择一行或多行数据：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com//titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)

# 选择第一行
dataframe.iloc[0]
```

```
Name        Allen, Miss Elisabeth Walton
PClass                               1st
Age                                   29
Sex                               female
Survived                               1
SexCode                                1
Name: 0, dtype: object
```

可以使用冒号“:”来定义想要选择哪些行。比如选择第 2、3、4 行：

```
# 选择三行
dataframe.iloc[1:4]
```

我们甚至还能获取到某一行为止的所有行，比如获取到第 4 行为止的所有行：

```
# 获取到第 4 行为止的所有行
dataframe.iloc[:4]
```

	Name	PClass	Age	Sex	Survived	SexCode
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.0	female	0	1

数据帧的索引不必非得是数值型。只要某一列在数据帧中每一行的值是唯一的，就可以将其设置为索引。举个例子，我们可以将乘客的名字设置为索引，然后通过名字来选择行：

```
# 设置索引
dataframe = dataframe.set_index(dataframe['Name'])

# 查看行
dataframe.loc['Allen, Miss Elisabeth Walton']
```

```
Name        Allen, Miss Elisabeth Walton
PClass                               1st
Age                                   29
Sex                               female
Survived                               1
SexCode                                1
Name: Allen, Miss Elisabeth Walton, dtype: object
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.0	female	0	1

讨论

pandas 的数据帧中所有的行都会有一个唯一的索引值。默认情况下，这个索引是一个整数，它标明了这一行在数据帧中的行的位置。然而，索引不一定必须是这样的一个整数。数据帧的索引可以被设置成一个唯一的字母与数字组成的字符串或自定义数字。为了能选择一行或者部分行，pandas 提供了两个方法：

- 当数据帧的索引是一个标签时（例如，一个字符串），`loc` 比较常用。
- `iloc` 并不是根据索引来查找数据的，而是根据行号来查找的，行号从 `0` 开始，逐次加 `1`。举个例子，不管索引是一个整数还是一个标签，`iloc[0]` 都将返回第 1 行的数据。

由于在数据清洗时会频繁使用 loc 和 iloc，所以有必要熟悉它们。

3.4 根据条件语句来选择行

问题描述

根据某个条件语句来选择数据帧的行数据。

解决方案

利用 pandas 很容易实现这个功能。例如，选择泰坦尼克号上所有女性乘客的数据：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)

# 展示 Sex 列的值是 female 的前两行
dataframe[dataframe['Sex'] == 'female'].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

我们一起来看这个解决方案是怎么做的。`dataframe['Sex'] == 'female'` 就是条件语句。我们在它的外面包了一层 `dataframe[]` 来告诉 pandas：“选择数据帧中所有 `dataFrame['Sex']` 的值是 `'female'` 的行数据”。

要同时使用多个条件语句也很简单。下面的代码就筛选出了所有年龄大于或等于 65 岁的女性乘客的行数据：

```
# 过滤行
dataframe[(dataframe['Sex'] == 'female') & (dataframe['Age'] >= 65)]
```

	Name	PClass	Age	Sex	Survived	SexCode
73	Crosby, Mrs Edward Gi ord (Catherine Elizabet...	1st	69.0	female	1	1

讨论

有条件地筛选数据是数据整理中最常见的任务之一。我们很少会遇到需要使用所有原始数据的场景，一般只使用一部分原始数据。举个例子，你可能只对某几个省的商店感兴趣，或者只对某个特定年龄段的病人记录感兴趣。

3.5 替换值

问题描述

替换数据帧中的一些值。

解决方案

用 pandas 的 replace 方法能很容易地找到并替换一些值。例如，用 "Woman" 来替换 Sex 列中所有的 "female"：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)

# 替换一些值，并查看两行数据
dataframe['Sex'].replace("female", "Woman").head(2)

0    Woman
1    Woman
Name: Sex, dtype: object
```

当然，我们也能同时替换多个值：

```
# 用 "Woman" 和 "Man" 分别替换 "female" 和 "male"
dataframe['Sex'].replace(["female","male"], ["Woman","Man"]).head(5)

0    Woman
1    Woman
2      Man
3    Woman
4      Man
```

```
Name: Sex, dtype: object
```

可以通过 Dataframe 对象在整个数据帧中查找和替换值，而不仅限于在单个列中查找和替换值：

```
# 替换一些值，并查看两行数据
dataframe.replace(1, "One").head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29	female	One	One
1	Allison, Miss Helen Loraine	1st	2	female	0	One

replace 也接受正则表达式：

```
# 替换一些值，并查看两行数据
dataframe.replace(r"1st", "First", regex=True).head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	First	29.0	female	1	1
1	Allison, Miss Helen Loraine	First	2.0	female	0	1

讨论

replace 是一个用来做值替换的工具，很简单，而且它还接受正则表达式，所以它的功能其实很强大。

3.6 重命名列

问题描述

重命名 pandas 数据帧中的列。

解决方案

用 rename 方法来重命名列：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'
```

```
# 加载数据
dataframe = pd.read_csv(url)

# 重命名列，查看两行数据
dataframe.rename(columns={'PClass': 'Passenger Class'}).head(2)
```

	Name	Passenger Class	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

注意，rename 方法可以传入一个字典作为参数。用字典可以同时改变多个列名：

```
# 重命名列，查看两行数据
dataframe.rename(columns={'PClass': 'Passenger Class', 'Sex': 'Gender'}).
head(2)
```

	Name	Passenger Class	Age	Gender	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

讨论

以字典作为 columns 参数使用 rename 来重命名列是我最喜欢的方式，因为它能同时为多个列重命名。如果想同时为所有的列重命名，下面这一小段代码会很有用。它以旧列名为键、空字符串为值，创建了一个字典：

```
# 加载库
import collections

# 创建字典
column_names = collections.defaultdict(str)

# 创建键
for name in dataframe.columns:
    column_names[name]

# 查看字典
column_names

defaultdict(str,
            {'Age': '',
             'Name': '',
             'PClass': '',
             'Sex': '',
             'SexCode': '',
             'Survived': ''})
```

3.7 计算最小值、最大值、总和、平均值与计数值

问题描述

计算一个数值列的最小值、最大值、总和、平均值与计数值。

解决方案

pandas 提供了一些内置的方法来计算常见的描述性统计量：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)

# 计算描述统计量
print('Maximum:', dataframe['Age'].max())
print('Minimum:', dataframe['Age'].min())
print('Mean:', dataframe['Age'].mean())
print('Sum:', dataframe['Age'].sum())
print('Count:', dataframe['Age'].count())

Maximum: 71.0
Minimum: 0.17
Mean: 30.397989417989415
Sum: 22980.879999999997
Count: 756
```

讨论

除了解决方案中用到的描述性统计量，pandas 还提供了计算方差（`var`）、标准差（`std`）、峰态（`kurt`）、偏态（`skew`）、平均值标准误差（`sem`）、众数（`mode`）、中位数（`median`）以及很多其他描述性统计量的方法。

此外，也可以对整个数据帧应用这些方法：

```
# 查看计数
dataframe.count()

Name        1313
```

```
PClass      1313
Age          756
Sex         1313
Survived    1313
SexCode     1313
dtype: int64
```

3.8 查找唯一值

问题描述

找出一列中所有的唯一值。

解决方案

使用 unique 来查看由某一列中全部的唯一值组成的数组：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)

# 筛选出唯一值
dataframe['Sex'].unique()

array(['female', 'male'], dtype=object)
```

还可以使用 value_counts，它会显示所有的唯一值以及它们出现的次数：

```
# 查看计数
dataframe['Sex'].value_counts()

male      851
female    462
Name: Sex, dtype: int64
```

讨论

unique 和 value_counts 都可以用来处理和探索分类型数据的（categorical）列。在数据整理阶段，分类型数据的列中经常会有一些分类需要处理。例如，在泰坦尼克号的数据集中，PClass 这个列是用来表示乘客船票级别的。泰坦尼克号中有三种级别的船票，而如果使用 value_counts，我们就能看出数据中的一个问题：

```
# 查看计数
dataframe['PClass'].value_counts()

3rd 711
1st 322
2nd 279
*1
Name: PClass, dtype: int64
```

如我们所设想的那样，几乎所有乘客的船票都在这三种级别中，但有一位乘客的船票级别是 *。有很多种策略能处理这种类型的问题，在第 5 章会详细介绍。现在，我们只需要认识到“异常”类型的数据在分类型数据中很常见，不应该忽视它们。

最后，如果你只是想统计有多少个唯一值，则可以使用 nunique：

```
# 查看唯一值的个数
dataframe['PClass'].nunique()

4
```

3.9 处理缺失值

问题描述

筛选出数据帧中的缺失值。

解决方案

isnull 和 notnull 都能返回布尔型的值来表示一个值是否缺失：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'
```

```
# 加载数据
dataframe = pd.read_csv(url)

# 筛选出缺失值，查看两行
dataframe[dataframe['Age'].isnull()].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
12	Aubert, Mrs Leontine Pauline	1st	NaN	female	1	1
13	Barkworth, Mr Algernon H	1st	NaN	male	1	0

讨论

在数据整理中，缺失值是很常见的问题，很多人都低估了处理缺失值的难度。pandas 使用 NumPy 的 NaN（“Not A Number”，意为“不是一个数字”）来表示缺失值。但值得注意的是，pandas 没有实现 NaN。例如，如果想要将所有包含 male 的字符串替换为缺失值的话，就会得到一条错误消息：

```
# 尝试用 NaN 去替换值
dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)

---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-7-5682d714f87d> in <module>()
      1 # 尝试用 NaN 去替换值
----> 2 dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)



NameError: name 'NaN' is not defined
---------------------------------------------------------------------------
```

要想使用 NaN 就需要先导入 NumPy 库：

```
# 加载库
import numpy as np

# 用 NaN 替换一些值
dataframe['Sex'] = dataframe['Sex'].replace('male', np.nan)
```

有时候一个数据集会使用特殊的值来表示缺失的观察值，比如 NONE、-999 或者 .。pandas 的 read_csv 中有一个参数，允许用户指定一个值来代表缺失值：

```
# 加载数据，设置缺失值
dataframe = pd.read_csv(url, na_values=[np.nan, 'NONE', -999])
```

3.10 删除一列

问题描述

从数据帧中删除一列。

解决方案

要删除一列，最好的方式是使用 drop 方法，并传入参数 axis=1（即坐标轴列）：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)

# 删除列
dataframe.drop('Age', axis=1).head(2)
```

	Name	PClass	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	female	1	1
1	Allison, Miss Helen Loraine	1st	female	0	1

也可以使用一组列名作为参数一次删除多列：

```
# 删除多列
dataframe.drop(['Age', 'Sex'], axis=1).head(2)
```

	Name	PClass	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	1	1
1	Allison, Miss Helen Loraine	1st	0	1

如果某一列没有名字（常常会发生这种情况），可以使用 dataframe.columns 通过指定列下标的方式删除该列：

```
# 删除一列
dataframe.drop(dataframe.columns[1], axis=1).head(2)
```

	Name	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	29.0	female	1	1
1	Allison, Miss Helen Loraine	2.0	female	0	1

讨论

drop 是删除一列的常用方法。还有一种方法是 `del dataframe['Age']`，大部分情况下，它都能很好地工作，但考虑到它在 pandas 内部的调用方式（对此细节的讨论超出了本书的范围），一般不推荐使用。

最好养成一个习惯，永远不使用 pandas 的 `inplace=True` 参数。很多 pandas 的方法都包含一个 `inplace` 参数，当其被设置为 `True` 时，会直接修改数据帧本身。在对数据进行一系列更复杂的处理时，这可能会产生一些问题，因为数据帧会被视为可变对象（严格来说确实是这样的）。我推荐将数据帧视为不可变对象。例如：

```
 # 创建一个新的数据帧
dataframe_name_dropped = dataframe.drop(dataframe.columns[0], axis=1)
```

在这个例子中，我们没有改变 `dataframe`，而是返回了一个新的数据帧对象，这个对象是 `dataframe` 修改后的版本 `dataframe_name_dropped`。如果将数据帧视为不可变对象，会省去很多麻烦。

3.11 删除一行

问题描述

从数据帧中删除一行或多行数据。

解决方案

创建一个新的数据帧并通过布尔条件剔除你想删除的行：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)
```

```
# 删除一些行，查看输出结果的前两行
dataframe[dataframe['Sex'] != 'male'].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

讨论

平心而论，使用 drop 方法是个不错的选择（比如，使用 `df.drop([0, 1], axis=0` 来删除前两行），而更实用的方式是在 `df[]` 中插入布尔条件，因为可以利用条件语句来删除一行或一次删除多行（这种情况更常见）。

使用布尔条件通过匹配唯一值的方式可以轻松删除一行：

```
# 删除一行，查看输出结果的前两行
dataframe[dataframe['Name'] != 'Allison, Miss Helen Loraine'].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

甚至可以通过这种方式根据行的下标来删除一行：

```
# 删除一行，查看输出结果的前两行
dataframe[dataframe.index != 0].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

3.12 删除重复行

问题描述

删除数据帧中重复的行。

解决方案

使用 `drop_duplicates`，但是需要留意它的参数：

```
# 加载库
import pandas as pd
```

```
# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)

# 删除重复行，查看输出结果的前两行
dataframe.drop_duplicates().head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

讨论

细心的读者会注意到，上述解决方案实际上没有删除任何行：

```
# 查看行数
print("Number Of Rows In The Original DataFrame:", len(dataframe))
print("Number Of Rows After Deduping:", len(dataframe.drop_duplicates()))

Number Of Rows In The Original DataFrame: 1313
Number Of Rows After Deduping: 1313
```

这是因为 `drop_duplicates` 默认只删除那些所有列都完美匹配的行。在这种情况下，`dataframe` 中的每一行实际上都是唯一的。但只用某些列来检查重复行会是一个很常见的需求，好在使用 `subset` 参数就能实现：

```
# 删除重复行
dataframe.drop_duplicates(subset=['Sex'])
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

仔细看看之前的输出：`drop_duplicates` 仅在 `Sex` 这一列有两行的值相同时，将重复的行删除。现在数据帧只剩下了两行：一行为男性，一行为女性。你可能会问，为什么 `drop_duplicates` 决定保留这两行而不是其他行？答案是 `drop_duplicates` 默认保留重复行中先出现的行，然后将剩余的行删除。可以通过 `keep` 参数来改变这个行为：

```
# 删除重复行
dataframe.drop_duplicates(subset=['Sex'], keep='last')
```

	Name	PClass	Age	Sex	Survived	SexCode
1307	Zabour, Miss Tamini	3rd	NaN	female	0	1
1312	Zimmerman, Leo	3rd	29.0	male	0	0

还有一个相关的方法是 duplicated，它返回一个布尔序列来表示某一行是否为重复的行。如果你不想简单地将重复行删除，这个方法会是一个不错的选择。

3.13 根据值对行分组

问题描述

要根据一些共有的值（shared value）对行分组。

解决方案

groupby 是 pandas 中最强大的功能之一：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)

# 根据 Sex 列的值来对行分组，并计算每一组的平均值
dataframe.groupby('Sex').mean()
```

Sex	Age	Survived	SexCode
female	29.396424	0.666667	1.0
male	31.014338	0.166863	0.0

讨论

groupby 才是数据整理工作的真正起点。经常会遇到这种情况：数据帧的每一行代表的是一个人或者一个事件，而我们需要根据某些标准对这些行分组并计算某个统计量。例如，假设有一个数据帧，其中每一行都是一家国际连锁餐厅分店的一条销售记录。如果想要知道每一家分店的销售总额，可以将销售记录按照分店分组并计算每一组的总和。

不熟悉 groupby 的用户经常会写一行下面的语句，然后就被返回的值弄晕了：

```
# 对行进行分组
dataframe.groupby('Sex')

<pandas.core.groupby.DataFrameGroupBy object at 0x10efacf28>
```

为什么不返回一些更有用的东西？原因是 groupby 需要和一些作用于组（group）的操作配合使用，比如计算一个综合统计量（例如，平均值、中位数、总和）。当说到分组时，我们总是用“按照性别分组”这样的简单说法，其实这是不完整的。为了让分组能更有用，我们需要根据某种标准进行分组，然后对每一组应用一个函数：

```
# 按行分组，计算行数
dataframe.groupby( 'Survived' )[ 'Name' ].count()

Survived
0    863
1    450
Name: Name, dtype: int64
```

注意到 Name 被加到 groupby 的后面了吗？这是因为特定的描述性统计量只对特定的数据类型来说才有意义。比如，按照性别来计算平均年龄是有意义的，但是按照性别来计算总年龄就没有什么意义。在上面这个例子中，我们先按照乘客是否幸存做了分组，然后计算了每一组名字的数量（也就是乘客的数量）。

当然，也可以先按照第一列分组，再按照第二列对前面的结果进行二次分组：

```
# 对行分组，计算平均值
dataframe.groupby(['Sex','Survived'])['Age'].mean()


Sex     Survived
female  0           24.901408
        1           30.867143
male    0           32.320780
        1           25.951875
Name: Age, dtype: float64
```

3.14 按时间段对行分组

问题描述

要按照时间段对行进行分组。

解决方案

使用 resample 按照时间段对行分组：

```
# 加载库
import pandas as pd
import numpy as np

# 创建日期范围
time_index = pd.date_range('06/06/2017', periods=100000, freq='30S')

# 创建数据帧
dataframe = pd.DataFrame(index=time_index)

# 创建一列随机变量
dataframe['Sale_Amount'] = np.random.randint(1, 10, 100000)

# 按周对行分组，计算每一周的总和
dataframe.resample('W').sum()
```

	Sale_Amount
2017-06-11	86423
2017-06-18	101045
2017-06-25	100867
2017-07-02	100894
2017-07-09	100438
2017-07-16	10297

讨论

标准的泰坦尼克号数据集中没有包含日期值的列，所以本节生成了一个简单的数据帧，每一行都是一条销售记录。对于每一条销售记录，日期、时间以及金额都是已知的（这个数据集中的数据是虚构的，因为每一条销售记录都刚好相隔 30 秒，并且金额刚好是整数。但是为了便于讨论，就姑且认为这是一个真实的数据集吧）。

原始数据看起来是这样的：

```
# 查看 3 行
dataframe.head(3)
```

	Sale_Amount
2017-06-11 00:00:00	7
2017-06-11 00:00:30	2
2017-06-11 00:01:00	7

注意，每一条销售记录的日期和时间都是数据帧的索引，这是因为 resample 要求索引的类型必须是类 datetime 的值。

使用 resample 可以按一组时间间隔（偏移）来对行分组，然后计算每一个时间组的某个统计量：

```
# 按两周分组，计算平均值
dataframe.resample('2W').mean()
```

	Sale_Amount
2017-06-11	5.001331
2017-06-25	5.007738
2017-07-09	4.993353
2017-07-23	4.950481

```
# 按月分组，计算行数
dataframe.resample('M').count()
```

	Sale_Amount
2017-06-30	72000
2017-07-31	28000

你可能已经注意到，尽管上面两个方法分别按周和按月对行分组，但输出的结果都是以日期作为索引的。这是因为在默认情况下，resample 会返回时间组的右边界的值（最后一个标签），作为这个组的标签。可以通过使用 label 参数改变这个行为：

```
# 按月分组，计算行数
dataframe.resample('M', label='left').count()
```

	Sale_Amount
2017-05-31	72000
2017-06-30	28000

延伸阅读

- pandas 的时间偏移别名列表（*http://bit.ly/2FxfTe4*）

3.15 遍历一个列的数据

问题描述

遍历一个列中的所有元素，并做某个操作。

解决方案

将 pandas 的列视为与 Python 中的其他序列一样：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)

# 以大写的形式打印前两行中的名字
for name in dataframe['Name'][0:2]:
    print(name.upper())

ALLEN, MISS ELISABETH WALTON
ALLISON, MISS HELEN LORAINE
```

讨论

除了循环（常称为 `for` 循环），我们还可以使用列表解析式：

```
# 以大写的形式打印前两行的名字
[name.upper() for name in dataframe['Name'][0:2]]

['ALLEN, MISS ELISABETH WALTON', 'ALLISON, MISS HELEN LORAINE']
```

尽管 `for` 循环很诱人，但是更具 Python 风格的解决方案是使用 pandas 的 `apply` 方法，我们将在 3.16 节详细说明。

3.16 对一列的所有元素应用某个函数

问题描述

在一列的所有元素上应用某个函数。

解决方案

使用 `apply` 对一列的所有元素应用一个内置的或者自定义的函数：

```
# 加载库
import pandas as pd
```

```
# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)

# 创建一个函数
def uppercase(x):
    return x.upper()

# 应用函数，查看两行
dataframe['Name'].apply(uppercase)[0:2]

0    ALLEN, MISS ELISABETH WALTON
1    ALLISON, MISS HELEN LORAINE
Name: Name, dtype: object
```

讨论

在数据清洗和数据整理中，`apply`是一个功能非常强大的函数。通常需要先写一个函数来执行某个有用的操作（例如，将姓和名分开、将字符串转换成浮点数等），然后对一列中的所有元素应用这个函数。

3.17 对所有分组应用一个函数

问题描述

我们已经使用 groupby 对行做了分组，现在要对每一组数据应用一个函数。

解决方案

联合使用 groupby 和 apply：

```
# 加载库
import pandas as pd

# 创建 URL
url = 'https://tinyurl.com/titanic-csv'

# 加载数据
dataframe = pd.read_csv(url)
```

```
# 对行分组，然后在每一组上应用函数
dataframe.groupby('Sex').apply(lambda x: x.count())
```

Sex	Name	PClass	Age	Sex	Survived	SexCode
female	462	462	288	462	462	462
male	851	851	468	851	851	851

讨论

3.16 节提到过 `apply`，当需要对所有分组应用一个函数的时候，`apply` 非常有用。通过联合使用 `groupby` 和 `apply`，我们就能计算自定义的统计量，也能对每一组数据分别应用任何函数。

3.18 连接多个数据帧

问题描述

连接（concatenate）两个数据帧。

解决方案

使用 `concat` 并设置参数 `axis=0`，沿着行的方向来连接数据帧：

```
# 加载库
import pandas as pd

# 创建数据帧
data_a = {'id': ['1', '2', '3'],
          'first': ['Alex', 'Amy', 'Allen'],
          'last': ['Anderson', 'Ackerman', 'Ali']}
dataframe_a = pd.DataFrame(data_a, columns = ['id', 'first', 'last'])

# 创建数据帧
data_b = {'id': ['4', '5', '6'],
          'first': ['Billy', 'Brian', 'Bran'],
          'last': ['Bonder', 'Black', 'Balwner']}
dataframe_b = pd.DataFrame(data_b, columns = ['id', 'first', 'last'])

# 沿着行的方向连接两个数据帧
pd.concat([dataframe_a, dataframe_b], axis=0)
```

	id	first	last
0	1	Alex	Anderson
1	2	Amy	Ackerman
2	3	Allen	Ali
0	4	Billy	Bonder
1	5	Brian	Black
2	6	Bran	Balwner

使用 axis=1 可以在列的方向上做连接：

```
# 在列的方向上连接两个数据帧
pd.concat([dataframe_a, dataframe_b], axis=1)
```

	id	first	last	id	first	last
0	1	Alex	Anderson	4	Billy	Bonder
1	2	Amy	Ackerman	5	Brian	Black
2	3	Allen	Ali	6	Bran	Balwner

讨论

在计算机科学和编程的世界之外，你很少会听说“连接（concatenate）”这个词，所以如果你之前没有听过，也不需要担心。*concatenate* 的官方定义是将两个对象黏在一起。在我们的解决方案中，两个小的数据帧被黏在一起，并且 axis 参数会指出，是将它们叠在一起还是并列排放。

append 也能将一行新数据附加（append）到数据帧中：

```
# 创建一行
row = pd.Series([10, 'Chris', 'Chillon'], index=['id', 'first', 'last'])

# 附加一行
dataframe_a.append(row, ignore_index=True)
```

	id	first	last
0	1	Alex	Anderson
1	2	Amy	Ackerman
2	3	Allen	Ali
3	10	Chris	Chillon

3.19 合并两个数据帧

问题描述

合并两个数据帧。

解决方案

要进行等值连接（inner join），就需要使用 merge 并用 on 参数来指定哪些列要合并：

```
# 加载库
import pandas as pd

# 创建 DataFrame
employee_data = {'employee_id': ['1', '2', '3', '4'],
                 'name': ['Amy Jones', 'Allen Keys', 'Alice Bees',
                 'Tim Horton']}
dataframe_employees = pd.DataFrame(employee_data, columns = ['employee_id',
                                                             'name'])

# 创建 DataFrame
sales_data = {'employee_id': ['3', '4', '5', '6'],
              'total_sales': [23456, 2512, 2345, 1455]}
dataframe_sales = pd.DataFrame(sales_data, columns = ['employee_id',
                                                      'total_sales'])

# 合并数据帧
pd.merge(dataframe_employees, dataframe_sales, on='employee_id')
```

	employee_id	name	total_sales
0	3	Alex Bees	23456
1	4	Tim Horton	2512

merge 默认进行等值连接。如果要进行外连接（outer join），可以通过 how 参数来指定：

```
# 合并两个数据帧
pd.merge(dataframe_employees, dataframe_sales, on='employee_id', how='outer')
```

	employee_id	name	total_sales
0	1	Amy Jones	NaN
1	2	Allen Keys	NaN
2	3	Alice Bees	23456.0
3	4	Tim Horton	2512.0
4	5	NaN	2345.0
5	6	NaN	1455.0

使用 how 参数还可以指定是左连接还是右连接：

```
# 合并两个数据帧
pd.merge(dataframe_employees, dataframe_sales, on='employee_id', how='left')
```

	employee_id	name	total_sales
0	1	Amy Jones	NaN
1	2	Allen Keys	NaN
2	3	Alice Bees	23456.0
3	4	Tim Horton	2512.0

也可以指定每个数据帧中的列名来进行合并：

```
# 合并两个数据帧
pd.merge(dataframe_employees,
         dataframe_sales,
         left_on='employee_id',
         right_on='employee_id')
```

	employee_id	name	total_sales
0	3	Alice Bees	23456
1	4	Tim Horton	2512

如果想根据数据帧的索引合并两列，可以将 `left_on` 和 `right_on` 参数替换成 `right_index=True` 以及 `left_index=True`。

讨论

很多时候我们需要使用的数据是很复杂的，而且它们并不总是完整的。其实在真实场景中，数据集通常是分散的，它们来自对数据库的多次查询或者多个文件。所以，为了将所有数据放在一起，通常需要先将每一份从数据库查询得到的数据或文件作为数据帧加载到 pandas 中，然后再将它们合并为一个大的数据帧。

这个过程对于用过 SQL（一个常用来做数据合并操作（join）的语言）的人来说可能很熟悉。尽管在 pandas 中使用的具体参数不一样，但是它所遵循的大致模式与其他语言和工具没有什么区别。

对于任何 `merge` 操作，都需要说明三个方面的信息。第一，必须指定想要合并哪两个数据帧。在上述解决方案中，它们是 `dataframe_employees` 和 `dataframe_sales`。第二，必须指定要根据哪两列实施合并，也就是说在两个数据帧中需要共享哪一列的值。比如，在上述解决方案中，两个数据帧都有一列叫作 `employee_id`，为了将这两个数据帧合并，只需要找到每个数据帧的 `employee_id` 列中能匹配的值，并把它们拼接起来就行。如果

这两列使用相同的名字，可以使用 on 参数。如果它们的名字不一样，则需要使用 left_on 和 right_on。

左 / 右数据帧是怎么定义的呢？简单来说，左数据帧是在 merge 中指定的第一个数据帧，右数据帧就是第二个。在我们解释下一组参数时，还会用到左 / 右数据帧的概念。

最后一个参数，也是大多数人都很难理解的参数，是指合并操作的类型，由 how 参数来指定。merge 支持如下 4 个主要的连接类型。

Inner

只返回指定列的值在两个数据帧中都存在的行。例如，只有 employee_id 的值在 dataframe_employees 和 dataframe_sales 中都出现时，才会返回行。

Outer

返回两个数据帧的所有行。如果某一行只在其中一个数据帧中存在，就用 NaN 来填充缺失的值。例如，返回 dataframe_employees 和 dataframe_sales 的所有行。

Left

返回左数据帧的所有行，对于右数据帧，只返回在左数据帧中能找到匹配值的行。用 NaN 来填充缺失的值。例如，返回 dataframe_employees 的所有行，但是对于 dataframe_sales，只返回它的 employee_id 出现在 dataframe_employees 中的那些行。

Right

返回右数据帧的所有行，对于左数据帧，只返回在右数据帧中能找到匹配值的行。用 NaN 来填充缺失的值。例如，返回 dataframe_sales 的所有行，但是对于 dataframe_employees，只返回它的 employee_id 出现在 dataframe_sales 中的那些行。

如果你无法一下子理解上述所有概念，推荐你自己动手写代码调试一下 how 参数，看一看它是如何影响 merge 的输出的。

延伸阅读

- 对 SQL Join 操作的可视化解释（*http://bit.ly/2Fxgcpe*）
- pandas 中关于合并操作的文档（*http://bit.ly/2Fuo4rH*）

第 4 章

处理数值型数据

4.0 简介

定量的数据是用来度量某个对象的，比如一个班有多少人、每月的销售额或者学生的成绩等。这些量的最自然的表示方式就是数字（例如，一个班有 29 个学生，销售额是 529,392 美元）。本章会讲解多种将原始数值型数据转换成机器学习算法所需特征的方法。

4.1 特征的缩放

问题描述

将一个数值型特征的值缩放（rescale）到两个特定的值之间。

解决方案

用 scikit-learn 的 `MinMaxScaler` 来缩放一个特征数组 ：

```
# 加载库
import numpy as np
from sklearn import preprocessing

# 创建特征
feature = np.array([[-500.5],
                    [-100.1],
                    [0],
                    [100.1],
                    [900.9]])
```

```
# 创建缩放器
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))

# 缩放特征的值
scaled_feature = minmax_scale.fit_transform(feature)

# 查看特征
scaled_feature

array([[ 0.        ],
       [ 0.28571429],
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

讨论

在机器学习中，缩放是一个很常见的预处理任务。本书后面所讲的很多算法都假设所有的特征是在同一取值范围中的，最常见的范围是 [0, 1] 或 [-1, 1]。用于缩放的方法有很多，其中最简单的一种被称为 min-max 缩放。min-max 缩放利用特征的最小值和最大值，将所有特征都缩放到同一个范围中。其计算公式如下：

$$x_i' = \frac{x_i - \min(\boldsymbol{x})}{\max(\boldsymbol{x}) - \min(\boldsymbol{x})}$$

$\boldsymbol{x}$ 是特征向量，x_i 是 $\boldsymbol{x}$ 中的一个元素值，x_i' 是缩放后的元素值。从本例输出的数组中可以看出，特征的元素值已经被成功地缩放到 0 和 1 之间了：

```
array([[ 0.        ],
       [ 0.28571429],
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

scikit-learn 的 `MinMaxScaler` 支持两种方式来缩放特征。第一种方式是使用 `fit` 计算特征的最小值和最大值，然后使用 `transform` 来缩放。第二种方式是使用 `fit_transform` 一次性执行上面所说的两个操作。从数学上来说，这两种方式没有区别。但是，有时候将两个操作分开会带来一些好处，因为这样的话我们就能对不同的数据集执行同样的转换。

延伸阅读

- 特征缩放（Wikipedia，*http://bit.ly/2Fuug2Z*）

- 《关于特征缩放和归一化》（Sebastian Raschka，*http://bit.ly/2FwwRcM*）

4.2 特征的标准化

问题描述

对一个特征进行转换，使其平均值为 0、标准差为 1。

解决方案

scikit-learn 的 `StandardScaler` 能同时执行这两个转换：

```
# 加载库
import numpy as np
from sklearn import preprocessing

# 创建特征
x = np.array([[-1000.1],
              [-200.2],
              [500.5],
              [600.6],
              [9000.9]])

# 创建缩放器
scaler = preprocessing.StandardScaler()

# 转换特征
standardized = scaler.fit_transform(x)

# 查看特征
standardized
array([[-0.76058269],
       [-0.54177196],
       [-0.35009716],
       [-0.32271504],
       [ 1.97516685]])
```

讨论

4.1 节中讨论的 min-max 缩放有一个常见的替代方案，就是将特征缩放为大致符合标准正态分布的。为了实现这样的缩放，我们使用标准化（standardization）方法来转换数据，这样数据就能有一个等于 0 的平均值 $\bar{x}$ 和一个等于 1 的标准差 σ。也就是说，特征中的每个元素都会被转换，使得：

$$x_i' = \frac{x_i - \bar{x}}{\sigma}$$

x_i'是 x_i 标准化后的形式。转换后的特征表示原始值距离平均值多少个标准差（在统计学中也称为 z 分数）。

标准化方法是机器学习数据预处理中的常用缩放方法，从我的经验来看，它比 min-max 缩放用得更多，但这也要看你所使用的机器学习算法。例如，在主成分分析中标准化方法更有用，而在神经网络中则更推荐使用 min-max 缩放（在本书的后面会详细讨论这两个算法）。一般来说，如果没有特殊原因，我推荐使用标准化方法进行缩放。

查看解决方案中最后输出的平均值和标准差，我们就能知道标准化的效果：

```
# 打印平均值和标准差
print("Mean:", round(standardized.mean()))
print("Standard deviation:", standardized.std())

Mean: 0.0
Standard deviation: 1.0
```

如果数据中存在很大的异常值，可能会影响特征的平均值和方差，也会对标准化的效果造成不良影响。在这种情况下，使用中位数和四分位数间距进行缩放会更有效。在 scikit-learn 中，具体的做法就是调用 `RobustScaler`：

```
# 创建缩放器
robust_scaler = preprocessing.RobustScaler()

# 转换特征
robust_scaler.fit_transform(x)

array([[ -1.87387612],
       [ -0.875     ],
       [  0.        ],
       [ 0.125     ],
       [ 10.61488511]])
```

4.3 归一化观察值

问题描述

对观察值的每一个特征进行缩放，使其拥有一致的范数（总长度是 1）。

解决方案

使用 Normalizer 并指定 norm 参数：

```
# 加载库
import numpy as np
from sklearn.preprocessing import Normalizer

# 创建特征矩阵
features = np.array([[0.5, 0.5],
                     [1.1, 3.4],
                     [1.5, 20.2],
                     [1.63, 34.4],
                     [10.9, 3.3]])

# 创建归一化器
normalizer = Normalizer(norm=" l2" )

# 转换特征矩阵
normalizer.transform(features)


array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

讨论

很多缩放方法（例如，min-max 缩放和标准化方法）是对特征进行操作的，但其实对观察值也可以进行缩放操作。Normalizer 可以对单个观察值进行缩放，使其拥有一致的范数（总长度为 1）。当一个观察值有多个相等的特征时（例如，做文本分类时，每一个词或每几个词就是一个特征），经常使用这种类型的缩放。

Normalizer 提供三个范数选项，默认值是欧氏范数（Euclidean norm，常被称为 L2 范数）：

$$\| x \|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

其中，x 是一个独立观察值，x_n 是这个观察值的第 n 个特征。

```
# 转换特征矩阵
features_l2_norm = Normalizer(norm=" l2" ).transform(features)
```

```
# 查看特征矩阵
features_l2_norm
```

```
array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

或者，可以指定使用曼哈顿范数（L1 范数）：

$$\| x \|_1 = \sum_{i=1}^{n} |x_i|$$

```
# 转换特征矩阵
features_l1_norm = Normalizer(norm="l1").transform(features)
```

```
# 查看特征矩阵
features_l1_norm
```

```
array([[ 0.5       ,  0.5       ],
       [ 0.24444444,  0.75555556],
       [ 0.06912442,  0.93087558],
       [ 0.04524008,  0.95475992],
       [ 0.76760563,  0.23239437]])
```

直观地说，L2 范数可以被视为纽约市中两个点之间的距离（也就是直线距离），而 L1 范数可以被视为一个人沿着街道行走的距离（向北走一个街区，向东走一个街区，然后向北走一个街区，再向东走一个街区，等等），这就是为什么它被称为“曼哈顿范数”或者“出租车范数”的原因。

实际上，当你用 `norm='l1'` 对一个观察值进行缩放后，它的元素总和为 1。有时候这个特性很有用：

```
# 查看总和
print("Sum of the first observation\'s values:",
   features_l1_norm[0, 0] + features_l1_norm[0, 1])
```

```
Sum of the first observation's values: 1.0
```

4.4 生成多项式和交互特征

问题描述

创建多项式特征和交互特征。

解决方案

虽然有些人选择自己手动创建多项式特征和交互特征，但其实 scikit-learn 提供了一个内置的方法：

```
# 加载库
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# 创建特征矩阵
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]]),
# 创建 PolynomialFeatures 对象
polynomial_interaction = PolynomialFeatures(degree=2, include_bias=False)

# 创建多项式特征
polynomial_interaction.fit_transform(features)

array([[ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.]])
```

degree 参数决定了多项式的最高阶数。例如，degree=2 会创建阶数最高为 2 的特征：

$$x_1,\ x_2,\ x_1^2,\ x_2^2$$

degree=3 则会创建阶数最高为 3 的特征：

$$x_1,\ x_2,\ x_1^2,\ x_2^2,\ x_1^3,\ x_2^3$$

此外，默认情况下 PolynomialFeatures 包含交互特征：

$$x_1 x_2$$

通过设置 interaction_only 为 True，可以强制创建出来的特征只包含交互特征：

```
interaction = PolynomialFeatures(degree=2,
              interaction_only=True, include_bias=False)
              interaction.fit_transform(features)

array([[ 2.,  3.,  6.],
       [ 2.,  3.,  6.],
       [ 2.,  3.,  6.]])
```

讨论

当特征和目标值（预测值）之间存在非线性关系时，就需要创建多项式特征。例如，年龄可能和身体状况有很大的关系，一般年龄越大身体状况越差，但是它们之间的关系不是线性关系，所以需要对特征 x 编码——生成这个特征的更高阶的形式（x^2、x^3 等）——以此表示对目标值造成的非线性影响。

另外，有时我们会遇到一个特征需要依赖于另一个特征才能对目标值造成影响的情况。举一个简单的例子，要预测一杯咖啡是否是甜的，要考虑两个特征：咖啡是否被搅拌过；咖啡是否加了糖。单独看二者中的任何一个，都无法确定这杯咖啡是甜的，但是把它们结合在一起，就能确定了。咖啡只有在加糖并且搅拌后才是甜的，这两个特征对目标值（咖啡是甜的）的作用是相互依赖的。生成一个交互特征（将两个特征相乘），我们就可以为这种关系编码。

4.5 转换特征

问题描述

对一个或多个特征进行自定义的转换。

解决方案

在 scikit-learn 中，使用 FunctionTransformer 对一组特征应用一个函数：

```
# 加载库
import numpy as np
from sklearn.preprocessing import FunctionTransformer

# 创建特征矩阵
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])
```

```
# 定义一个简单的函数
def add_ten(x):
    return x + 10

# 创建转换器
ten_transformer = FunctionTransformer(add_ten)

# 转换特征矩阵
ten_transformer.transform(features)

array([[12, 13],
       [12, 13],
       [12, 13]])
```

在 pandas 中可以使用 apply 进行同样的转换：

```
# 加载库
import pandas as pd

# 创建数据帧
df = pd.DataFrame(features, columns=[ “feature_1” , “feature_2” ])

# 应用函数
df.apply(add_ten)
```

	feature_1	feature_2
0	12	13
1	12	13
2	12	13

讨论

对一个或多个特征进行自定义的转换是很常见操作。例如，你可能想创建一个特征来表示另一个特征的自然对数值。创建一个函数，并使用 scikit-learn 的 FunctionTransformer 或 pandas 的 apply 来应用这个函数，就能做到这一点。上述解决方案中创建了一个很简单的函数 add_ten，它将每一个输入值加 10。当然，我们也完全可以定义一个复杂得多的函数。

4.6 识别异常值

问题描述

识别样本中的一些极端观察值（异常值）。

解决方案

与其说识别异常值（outlier）是一门技术，不如说它是一门艺术。常用的方法是假设数据是正态分布的，基于这个假设，在数据周围“画”一个椭圆，将所有处于椭圆内的观察值视为正常值（标注为 1），将所有处于椭圆外的观察值视为异常值（标注为 −1）：

```
# 加载库
import numpy as np
from sklearn.covariance import EllipticEnvelope
from sklearn.datasets import make_blobs

# 创建模拟数据
features, _ = make_blobs(n_samples = 10,
                         n_features = 2,
                         centers = 1,
                         random_state = 1)

# 将第一个观察值的值替换为极端值
features[0,0] = 10000
features[0,1] = 10000

# 创建识别器
outlier_detector = EllipticEnvelope(contamination=.1)

# 拟合识别器
outlier_detector.fit(features)

# 预测异常值
outlier_detector.predict(features)

array([-1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

这个方法的一个主要限制是它需要指定一个 `contamination`（污染指数）参数，表示异常值在观察值中的比例——这个值我们也不知道是多少。可以将 `contamination` 视为你估计的数据的清洁程度。如果你认为数据中只有很少几个异常值，可以将 `contamination` 设置得小一点。反之，如果数据中很有可能有好几个异常值，就将 `contamination` 设置为一个更大的值。

除了查看所有观察值，我们还可以只查看某些特征，并使用四分位差（interqutile range，IQR）来识别这些特征的极端值。

```
# 创建一个特征
feature = features[:,0]
```

```
# 创建一个函数来返回异常值的下标
def indicies_of_outliers(x):
    q1, q3 = np.percentile(x, [25, 75])
    iqr = q3 - q1
    lower_bound = q1 - (iqr * 1.5)
    upper_bound = q3 + (iqr * 1.5)
    return np.where((x > upper_bound) | (x < lower_bound))

# 执行函数
indicies_of_outliers(feature)

(array([0]),)
```

IQR 是数据集的第 1 个四分位数和第 3 个四分位数之差。可以将 IQR 视为数据集中大部分数据的延展距离，而异常值会远远地偏离数据较为集中的区域。异常值常常被定义为比第 1 个四分位数小 1.5 IQR（即 IQR 的 1.5 倍）的值，或比第 3 个四分位数大 1.5 IQR 的值。

讨论

实际上没有一个通用的识别异常值的解决方案。每一种技术都有它的优点和缺点。最好的策略是尝试多种技术（比如 `EllipticEnvelope` 和基于 IQR 的识别）并从整体上来看结果。

如果可能的话，你应该仔细看一看被识别为异常值的观察值，并尝试去解释它们。例如，假设你有一个关于房子的数据集，其中一个特征是房间数。如果一个房子拥有 100 个房间，那么这个房子真的是一个异常值，还是说它实际上是旅馆而被错误地划归为异常值了呢？

延伸阅读

- 三种识别异常值的方法（以及本书用到的 IQR 函数的源代码）（*http://bit.ly/2FzMC2k*）

4.7 处理异常值

问题描述

处理数据中的异常值。

解决方案

通常有三种方式来处理异常值。第一种方式，丢弃它们：

```
# 加载库
import pandas as pd

# 创建数据帧
houses = pd.DataFrame()
houses['Price'] = [534433, 392333, 293222, 4322032]
houses['Bathrooms'] = [2, 3.5, 2, 116]
houses['Square_Feet'] = [1500, 2500, 1500, 48000]

# 筛选观察值
houses[houses['Bathrooms'] < 20]
```

	Price	Bathrooms	Square_Feet
0	534433	2.0	1500
1	392333	3.5	2500
2	293222	2.0	1500

第二种方式，将它们标记为异常值，并作为数据的一个特征：

```
# 加载库
import numpy as np

# 基于布尔条件语句来创建特征
houses["Outlier"] = np.where(houses["Bathrooms"] < 20, 0, 1)

# 查看数据
houses
```

	Price	Bathrooms	Square_Feet	Outlier
0	534433	2.0	1500	0
1	392333	3.5	2500	0
2	293222	2.0	1500	0
3	4322032	116.0	48000	1

最后，对有异常值的特征进行转换，降低异常值的影响：

```
# 对特征取对数值
houses["Log_Of_Square_Feet"] = [np.log(x) for x in houses["Square_Feet"]]

# 查看数据
houses
```

	Price	Bathrooms	Square_Feet	Outlier	Log_Of_Sqaure_Feet
0	534433	2.0	1500	0	7.313220
1	392333	3.5	2500	0	7.824046
2	293222	2.0	1500	0	7.313220
3	4322032	116.0	48000	1	10.778956

讨论

和识别异常值一样，处理异常值时也不存在一个绝对准则。应该基于两个方面来考虑对异常值的处理。第一，要弄清楚是什么让它们成为异常值的。如果你认为它们是错误的观察值，比如它们来自一个坏掉的传感器或者是被记错了的值，那么就要丢弃它们或者用 `NaN` 来替换异常值，因为我们无法信任这些值。但是，如果你认为这些异常值真的就是极端值（例如一幢大宅子有 200 间卧室），那么把它们标记为异常值或者对它们的值进行转换，是更合理的做法。

第二，应该基于机器学习的目标来处理异常值。例如，如果想要基于房屋的特征来预测其价格，那么可以合理地假设有 100 间卧室的大宅子的价格是由不同于普通家庭住宅的特征驱动的。此外，如果使用一个在线住房贷款的 Web 应用的部分数据来训练一个模型，那么就要假设潜在用户中不存在想要买一栋有几百间卧室的豪宅的亿万富翁。

所以，对于异常值到底要如何处理呢？首先，想一想它们为什么是异常值，然后对于数据要有一个最终的目标。最重要的是，要记住“决定不处理异常值”本身就是一个有潜在影响的决定。

另外，如果数据中有异常值，那么采用标准化方法做缩放就不太合适了，因为平均值和方差受异常值的影响很大。这种情况下，需要针对异常值使用一个鲁棒性更高的缩放方法，比如 `RobustScaler`。

延伸阅读

- `RobustScaler` 文档（*http://bit.ly/2DcgyNT*）

4.8 将特征离散化

问题描述

将一个数值型特征离散化，分到多个离散的小区间中。

解决方案

根据数据离散化的方式，有两种方法可以使用。第一种方法是根据阈值将特征二值化：

```
# 加载库
import numpy as np
from sklearn.preprocessing import Binarizer

# 创建特征
age = np.array([[6],
                [12],
                [20],
                [36],
                [65]])

# 创建二值化器
binarizer = Binarizer(18)

# 转换特征
binarizer.fit_transform(age)

array([[0],
       [0],
       [1],
       [1],
       [1]])
```

第二种方法是根据多个阈值将数值型特征离散化：

```
# 将特征离散化
np.digitize(age, bins=[20,30,64])

array([[0],
       [0],
       [1],
       [2],
       [3]])
```

记住，`bins`参数中的每个数字表示的是每个区间的左边界（左闭右开）。例如，`20`表示第一个区间不包括值为 20 的元素，只包括两个值小于 20 的元素。可以设置`right`参数为 True 来改变这个行为：

```
# 将特征离散化
np.digitize(age, bins=[20,30,64], right=True)

array([[0],
```

```
       [0],
       [0],
       [2],
       [3]])
```

讨论

如果有足够的理由认为某个数值型特征应该被视为一个分类特征（categorical feature），那么离散化会是一个卓有成效的策略。例如，19 岁和 20 岁的人消费习惯差距很小，但是 20 岁和 21 岁（在美国，21 岁的年轻人就可以饮酒了）的人之间消费习惯差距会很大。在这个例子中，将人群划分成能喝酒的和不能喝酒的会很有用。同样，在其他情况下，将数据离散化为 3 个或更多区间也很有用。

在上述解决方案中，我们可以看到两种离散化方法：针对 2 个区间使用了 scikit-learn 的 `Binarizer`，针对 3 个及以上的区间使用了 NumPy 的 `digitize`。其实，只指定一个阈值，`digitize` 也能像 `Binarizer` 一样二值化特征：

```
# 将特征离散化
np.digitize(age, bins=[18])

array([[0],
       [0],
       [1],
       [1],
       [1]])
```

延伸阅读

- `digitize` 的文档（*http://bit.ly/2HSciFP*）

4.9 使用聚类的方式将观察值分组

问题描述

对观察值进行聚类操作，使相似的观察值被分为一组。

解决方案

如果你有 *k* 个分组，可以使用 K-Means（K 均值）聚类法将相似的观察值分到一个组，并输出一个新的特征，以标识观察值属于哪一组：

```
# 加载库
import pandas as pd
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# 创建模拟的特征矩阵
features, _ = make_blobs(n_samples = 50,
                         n_features = 2,
                         centers = 3,
                         random_state = 1)

# 创建数据帧
dataframe = pd.DataFrame(features, columns=["feature_1","feature_2"])

# 创建 K-Means 聚类器
clusterer = KMeans(3, random_state=0)

# 将聚类器应用在特征上
clusterer.fit(features)

# 预测聚类的值
dataframe["group"] = clusterer.predict(features)

# 查看前几个观察值
dataframe.head(5)
```

	feature_1	feature_2	group
0	-9.877553	-3.336145	0
1	-7.287210	-8.353986	2
2	-6.943061	-7.023744	2
3	-7.440167	-8.791959	2
4	-6.641388	-8.075888	2

讨论

在这里讨论聚类有点早了，本书在后面会更深入地介绍聚类算法。其实，这么早提到聚类主要是因为它可以作为预处理的一个步骤。明确地说，就是使用类似 K-Means 的无监督学习算法将观察值聚类成几组。最后得到一个分类特征，相似的观察值会被分到同一组。

如果你还看不懂这些也不要担心，只要记住在数据预处理中能用聚类就行。如果你已经迫不及待要了解聚类，可以先跳到第 19 章看一看。

4.10 删除带有缺失值的观察值

问题描述

删除带有缺失值的观察值。

解决方案

用一行 NumPy 代码就能轻松删除带缺失值的观察值：

```
# 加载库
import numpy as np

# 创建特征矩阵
features = np.array([[1.1, 11.1],
                     [2.2, 22.2],
                     [3.3, 33.3],
                     [4.4, 44.4],
                     [np.nan, 55]])

# 只保留没有（用~来表示）缺失值的观察值
features[~np.isnan(features).any(axis=1)]

array([[  1.1,  11.1],
       [  2.2,  22.2],
       [  3.3,  33.3],
       [  4.4,  44.4]])
```

或者，使用 pandas 丢弃有缺失值的观察值：

```
# 加载库
import pandas as pd

# 加载数据
dataframe = pd.DataFrame(features, columns=["feature_1","feature_2"])

# 删除带有缺失值的观察值
dataframe.dropna()
```

	feature_1	feature_2
0	1.1	11.1
1	2.2	22.2
2	3.3	33.3
3	4.4	44.4

讨论

大多数机器学习算法不允许目标值或特征数组中存在缺失值。因此，不能简单地忽略数据中的缺失值，而是要在数据预处理阶段解决这个问题。

最简单的解决方法是删除所有含缺失值的观察值，用 NumPy 或者 pandas 很容易实现。

即便如此，删除带缺失值的观察值也是一件令人心痛的决定。这样做会让算法丢失观察值中那些非缺失值的信息，所以删除观察值只能作为最终别无他法时不得已的选择。

还有一点很重要，删除观察值可能会在数据中引入偏差，这主要由缺失值的成因决定。缺失值一共有三种类型：

完全随机缺失（Missing Completely At Random，MCAR）

> 数据缺失的可能性与任何其他东西无关。例如，某个接受问卷调查的人会在回答问题前先掷一个骰子，如果掷出了 6，她就跳过那个问题不做回答。

随机缺失（Missing At Random，MAR）

> 数据缺失的可能性不是完全随机的，与已经存在的其他特征有关。例如，在一次问卷调查中会问及性别和薪资水平，那么接受调查的女性更可能会跳过薪资的问题，但是她们选择是否不作答要看我们是否已经得知其性别信息。

完全非随机缺失（Missing Not At Random，MNAR）

> 数据缺失的可能性完全是非随机的，并且与未在特征中反映出的信息有关。例如，一个问卷调查会问到被调查人的薪资水平，而女性更可能会跳过关于薪资水平的问题，但是我们的数据中没有关于性别的特征。

如果观察值是 MCAR 或者 MAR，那么有时候删除它们是可接受的。但是如果它们是 MNAR，那么数据缺失本身其实就是一个信息。删除 MNAR 观察值会导致数据产生偏差，因为这些观察值是由未观察到的系统效应产生的。

延伸阅读

- 《定义三种类型的缺失数据》（*http://bit.ly/2Fto4bx*）
- 《缺失数据的填充》（*http://bit.ly/2FAkKLI*）

4.11 填充缺失值

问题描述

数据中存在缺失值，我们希望填充或者预测这些值。

解决方案

如果数据量不大，可以使用 KNN（K-Nearest Neighbors，K 近邻）算法来预测缺失值：

```
# 加载库
import numpy as np
from fancyimpute import KNN
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

# 创建模拟特征矩阵
features, _ = make_blobs(n_samples = 1000,
                         n_features = 2,
                         random_state = 1)

# 标准化特征
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)

# 将第一个特征向量的第一个值替换为缺失值
true_value = standardized_features[0,0]
standardized_features[0,0] = np.nan

# 预测特征矩阵中的缺失值
features_knn_imputed = KNN(k=5, verbose=0).complete(standardized_features)

# 对比真实值和填充值
print("True Value:", true_value)
print("Imputed Value:", features_knn_imputed[0,0])

True Value: 0.8730186114
Imputed Value: 1.09553327131
```

还可以使用 scikit-learn 的 `Imputer` 模块，用特征的平均值、中位数或者众数来填充缺失值，不过效果通常都会比使用 KNN 的差：

```
# 加载库
from sklearn.preprocessing import Imputer
```

```
# 创建填充器
mean_imputer = Imputer(strategy="mean"sssssssssss, axis=0)

# 填充缺失值
features_mean_imputed = mean_imputer.fit_transform(features)

# 对比真实值和填充值
print("True Value:", true_value)
print("Imputed Value:", features_mean_imputed[0,0])

True Value: 0.8730186114
Imputed Value: -3.05837272461
```

讨论

填充缺失值的策略主要有两种，每种策略各有优缺点。首先，可以使用机器学习来预测缺失值。为了达到目的，可以将带有缺失值的特征当作一个目标向量，然后使用剩余的特征来预测缺失值。虽然可以使用各种机器学习算法来做预测，但是最流行的选择是KNN。我们会在第 14 章中详细介绍 KNN，这里仅做简单的讲解。作为一种机器学习算法，KNN 使用 *k* 个最近的观察值（根据某种距离度量方法计算出来的）来预测缺失值。上述解决方案中使用了 5 个最近的观察值来预测缺失值。

KNN 的不足是，为了知道哪些观察值离缺失值最近，需要计算每一个观察值与缺失值之间的距离。对于小数据集，这么做没有问题，但是如果数据集中有成千上万的观察值，计算量将成为一个很严重的问题。

一个比较容易扩展到大数据集的替代方案是用平均值来填充缺失值。例如，在我们的解决方案中，借助 scikit-learn 使用特征的平均值来填充缺失值。尽管这样做的效果通常没有使用 KNN 的好，但是“平均值填充策略”很容易扩展到包含成千上万观察值的大数据集。

如果要采用填充策略，最好创建一个二元特征来表明该观察值是否包含填充值。

延伸阅读

- 《关于将 KNN 作为填充方法的研究》（*http://bit.ly/2HS9sAT*）

第 5 章

处理分类数据

5.0 简介

有时候，根据某种特性而不是数量来度量对象会更有效。我们常常使用这种定性的信息来判断一个观察值的属性，比如按照性别、颜色或者车的品牌这样的类别对其分类。但并不是所有的分类数据都是这样的。本身没有内在顺序的类别称为 *nominal*，以下类别都属于 nominal 类别：

- 蓝色、红色、绿色
- 男性、女性
- 香蕉、草莓、苹果

相反，如果一组分类天然拥有内在的顺序性，它就被称为 *ordinal*，例如：

- 低、中、高
- 年轻、年迈
- 同意、中立、反对

此外，分类信息常常用一个字符串型的向量或列（例如，`"Maine"`、`"Texas"`、`"Delaware"`）表示，但大部分机器学习算法都要求其输入是数值型的数据。

KNN（K-Nearest Neighbor，K 近邻）算法就是一个简单的例子。该算法的第一步是计算两个观察值之间的距离，通常使用的是欧氏距离：

$$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

其中，x 和 y 是两个观察值，下标 i 表示这是观察值的第 i 个特征。如果 x_i 是一个字符串

（例如"Texas"），那么要计算距离显然是不可能的。因此，需要将字符串转换成某种数值形式，这样才能使用欧氏距离的计算公式。我们的目标是将分类数据中包含的信息合理地转换成数值（分类的序列数、两个分类之间的间距等）。本章会介绍多种转换方法，来应对在处理分类数据时经常碰到的一些问题。

5.1 对 nominal 型分类特征编码

问题描述

存在一个没有内部顺序的 nominal 型分类特征（例如，苹果、梨、香蕉），现在要对其进行处理。

解决方案

利用 scikit-learn 的 `LabelBinarizer` 对特征进行 one-hot 编码（独热编码）：

```
# 加载库
import numpy as np
from sklearn.preprocessing import LabelBinarizer, MultiLabelBinarizer

# 创建特征
feature = np.array([["Texas"],
                    ["California"],
                    ["Texas"],
                    ["Delaware"],
                    ["Texas"]])

# 创建 one-hot 编码器
one_hot = LabelBinarizer()

# 对特征进行 one-hot 编码
one_hot.fit_transform(feature)

array([[0, 0, 1],
       [1, 0, 0],
       [0, 0, 1],
       [0, 1, 0],
       [0, 0, 1]])
```

可以使用 `classes_` 方法输出分类：

```
# 查看特征的分类
one_hot.classes_
```

```
array(['California', 'Delaware', 'Texas'],
       dtype='<U10'
```

如果希望对 one-hot 编码进行逆转换，可以使用 inverse_transform：

```
# 对 one-hot 编码逆转换
one_hot.inverse_transform(one_hot.transform(feature))
array(['Texas', 'California', 'Texas', 'Delaware', 'Texas'],
       dtype='<U10')
```

我们甚至可以使用 pandas 对特征进行 one-hot 编码：

```
# 加载库
import pandas as pd

# 创建虚拟变量
pd.get_dummies(feature[:,0])
```

	California	Delaware	Texas
0	0	0	1
1	1	0	0
2	0	0	1
3	0	1	0
4	0	0	1

scikit-learn 还能处理每个观察值有多个分类的情况：

```
# 创建有多个分类的特征
multiclass_feature = [("Texas", "Florida"),
                      ("California","Alabama"),
                      ("Texas", "Florida"),
                      ("Delware", "Florida"),
                      ("Texas", "Alabama")]

# 创建能处理多个分类的 one-hot 编码器
one_hot_multiclass = MultiLabelBinarizer()

# 对特征进行 one-hot 编码
one_hot_multiclass.fit_transform(multiclass_feature)

array([[0, 0, 0, 1, 1],
       [1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1],
       [0, 0, 1, 1, 0],
       [1, 0, 0, 0, 1]])
```

当然，可以用 classes_ 方法查看分类：

```
# 查看分类
one_hot_multiclass.classes_

array(['Alabama', 'California', 'Delware', 'Florida', 'Texas'], dtype=object)
```

讨论

你可能认为给每个分类赋予一个数值型的值是一个很不错的方法（比如，Texas = 1，California = 2），但是由于这些分类没有内在的顺序（比如，实际上 Texas 并不比 California“小”），使用数值就会错误地赋予分类一个并不存在的顺序。

正确的策略是为原特征中的每一个分类都创建一个二元特征。这个方法被称为 one-hot 编码（在机器学习的文献中）或者虚拟变量（在统计学的文献中）。上述解决方案中的特征（向量）包含了三个分类（也就是 Texas、California、Delaware）。在 one-hot 编码中，每一个分类都被当作一个特征，如果特征存在就用 1 表示，否则用 0 表示。因为我们的特征有 3 个分类，所以 one-hot 编码会返回 3 个二元特征（每个分类有一个特征）。通过使用 one-hot 编码，我们能得到观察值在一个分类中的属性值，并且保留“分类没有层级（没有顺序）”的状态。

值得注意的是，在 one-hot 编码之后，最好从结果矩阵中删除一个 one-hot 编码的特征，以避免线性依赖。

延伸阅读

- 《虚拟变量陷阱》（Algosome，*http://bit.ly/2FvVJkC*）
- 使用 one-hot 编码的时候需要删除一列（CrossValidated，*http://bit.ly/2FwrxG0*）

5.2 对 ordinal 分类特征编码

问题描述

存在一个 ordinal 分类特征（例如，高、中等、低），现在要对其编码。

解决方案

使用 pandas 数据帧的 replace 方法将字符串标签转换成相应的数字：

```
# 加载库
```

```
import pandas as pd

# 创建特征
dataframe = pd.DataFrame({"Score": ["Low", "Low", "Medium", "Medium",
"High"]})

# 创建映射器
scale_mapper = {"Low":1,
                "Medium":2,
                "High":3}

# 使用映射器来替换特征
dataframe["Score"].replace(scale_mapper)

0   1
1   1
2   2
3   2
4   3
Name: Score, dtype: int64
```

讨论

有一些分类的特征总是有某种天然顺序的。李克特量表（Likert scale）就是一个典型的例子：

- 非常同意
- 同意
- 保持中立
- 反对
- 强烈反对

对那些用于机器学习的特征进行编码时，需要将 ordinal 分类转换成数值，同时保留其顺序。最常见的方法就是，创建一个字典，将分类的字符串标签映射为一个数字，然后将其映射在特征上。

有一点很关键，就是只有知道 ordinal 分类的顺序信息，才能决定每个分类对应的数值。在上述解决方案中，`high` 就是比 `low` 大三倍。这在绝大多数场景下是没有问题的，但是如果分类之间的间隔不相等，就有问题了：

```
dataframe = pd.DataFrame({"Score": ["Low",
                                    "Low",
                                    "Medium",
```

```
                                    "Medium",
                                    "High",
                                    "Barely More Than Medium"]})
scale_mapper = {"Low":1,
                "Medium":2,
                "Barely More Than Medium": 3,
                "High":4}


dataframe["Score"].replace(scale_mapper)

0    1
1    1
2    2
3    2
4    4
5    3
Name: Score, dtype: int64
```

在这个例子中，Low（低）和 Medium（中等）之间的距离与 Medium 和 Baraly More Than Medium（比中等稍微多一点点）之间的一样，这就不太准确了。最好多留意分类所映射的数值：

```
scale_mapper = {"Low":1,
                "Medium":2,
                "Barely More Than Medium": 2.1,
                "High":3}

dataframe["Score"].replace(scale_mapper)

0    1.0
1    1.0
2    2.0
3    2.0
4    3.0
5    2.1
Name: Score, dtype: float64
```

5.3 对特征字典编码

问题描述

将一个字典转换成一个特征矩阵。

解决方案

使用 DictVectorizer：

```
# 加载库
from sklearn.feature_extraction import DictVectorizer

# 创建一个字典
data_dict = [{"Red": 2, "Blue": 4},
             {"Red": 4, "Blue": 3},
             {"Red": 1, "Yellow": 2},
             {"Red": 2, "Yellow": 2}]

# 创建字典向量化器
dictvectorizer = DictVectorizer(sparse=False)

# 将字典转换成特征矩阵
features = dictvectorizer.fit_transform(data_dict)

# 查看特征矩阵
features

array([[ 4.,  2.,  0.],
       [ 3.,  4.,  0.],
       [ 0.,  1.,  2.],
       [ 0.,  2.,  2.]])
```

默认情况下，DictVectorizer 会输出一个稀疏矩阵来存储除 0 以外的元素。如果矩阵很庞大（在自然语言处理中，大矩阵是很常见的），这么做有助于节省内存。通过指定 sparse=False 能强制 DictVectorizer 输出一个稠密矩阵。

使用 get_feature_names 方法可以获取所生成的特征的名字：

```
# 获取特征的名字
feature_names = dictvectorizer.get_feature_names()

# 查看特征的名字
feature_names

['Blue', 'Red', 'Yellow']
```

尽管不是强制的，不过为了便于解释，我们使用 pandas 的数据帧，以便更清楚地查看输出：

```
# 加载库
import pandas as pd
```

```
# 从特征中创建数据帧
pd.DataFrame(features, columns=feature_names)
```

	Blue	Red	Yellow
0	4.0	2.0	0.0
1	3.0	4.0	0.0
2	0.0	1.0	2.0
3	0.0	2.0	2.0

讨论

在很多编程语言中，字典都是很流行的数据结构，但机器学习算法要求数据是矩阵的形式。还好我们使用 scikit-learn 的 dictvectorizer 就可以实现这样的转换。

在自然语言处理中，这样的场景很常见。例如，我们有大量文档，对每个文档都用一个字典存放每个词在文档中出现的次数。使用 dictvectorizer 可以很方便地创建一个特征矩阵，其中的每个特征都表示一个词在某个文档中出现的次数：

```
# 为 4 个文档创建词频字典
doc_1_word_count = {"Red": 2, "Blue": 4}
doc_2_word_count = {"Red": 4, "Blue": 3}
doc_3_word_count = {"Red": 1, "Yellow": 2}
doc_4_word_count = {"Red": 2, "Yellow": 2}

# 创建列表
doc_word_counts = [doc_1_word_count,
                   doc_2_word_count,
                   doc_3_word_count,
                   doc_4_word_count]

# 将词频字典列表转换成特征矩阵
dictvectorizer.fit_transform(doc_word_counts)

array([[ 4.,  2.,  0.],
       [ 3.,  4.,  0.],
       [ 0.,  1.,  2.],
       [ 0.,  2.,  2.]])
```

在玩具例子中，只有 3 个不同的词（Red、Yellow、Blue），所以特征矩阵中只有 3 个特征。但是，如果每个文档实际是图书馆中的一本书，那么可以想象特征矩阵将变得很庞大（那就需要将 sparse 设置为 True 了）。

延伸阅读

- 如何在 Python 中使用字典（*http://bit.ly/2HReoWz*）
- 《SciPy 中的稀疏矩阵》（*http://bit.ly/2HReBZR*）

5.4 填充缺失的分类值

问题描述

有一个分类特征中包含缺失值，需要用预测值来填充。

解决方案

最理想的解决方案是训练一个机器学习分类器来预测缺失值，通常会使用 KNN 分类器：

```
# 加载库
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

# 用分类特征创建特征矩阵
X = np.array([[0, 2.10, 1.45],
              [1, 1.18, 1.33],
              [0, 1.22, 1.27],
              [1, -0.21, -1.19]])

# 创建带缺失值的特征矩阵
X_with_nan = np.array([[np.nan, 0.87, 1.31],
                       [np.nan, -0.67, -0.22]])

# 训练 KNN 分类器
clf = KNeighborsClassifier(3, weights='distance')
trained_model = clf.fit(X[:,1:], X[:,0])

# 预测缺失值的分类
imputed_values = trained_model.predict(X_with_nan[:,1:])

# 将所预测的分类和它们的其他特征连接起来
X_with_imputed = np.hstack((imputed_values.reshape(-1,1), X_with_nan[:,1:]))

# 连接两个特征矩阵
np.vstack((X_with_imputed, X))
```

```
array([[ 0.  ,  0.87,  1.31],
       [ 1.  , -0.67, -0.22],
       [ 0.  ,  2.1 ,  1.45],
       [ 1.  ,  1.18,  1.33],
       [ 0.  ,  1.22,  1.27],
       [ 1.  , -0.21, -1.19]])
```

另一个解决方案是用特征中出现次数最多的值来填充缺失值：

```
from sklearn.preprocessing import Imputer

# 连接两个特征矩阵
X_complete = np.vstack((X_with_nan, X))

imputer = Imputer(strategy='most_frequent', axis=0)

imputer.fit_transform(X_complete)

array([[ 0.  ,  0.87,  1.31],
       [ 0.  , -0.67, -0.22],
       [ 0.  ,  2.1 ,  1.45],
       [ 1.  ,  1.18,  1.33],
       [ 0.  ,  1.22,  1.27],
       [ 1.  , -0.21, -1.19]])
```

讨论

当分类特征中存在缺失值的时候，最好的解决方案是利用机器学习算法预测缺失值。将带缺失值的特征作为目标向量，将其他特征作为特征矩阵，就能完成预测。常用的算法是 KNN（在本书的后面会更深入地讨论），它会将 k 个最近的观察值的中位数作为缺失值的填充值。

另外，可以用特征中出现次数最多的分类来填充缺失值。虽然比使用 KNN 效果差一些，但是它能更容易地扩展到大数据集上。不管是哪一种情况，最好都添加一个二元特征来标识观察值中是否包含填充值。

延伸阅读

- 《解决随机森林分类器中的缺失值问题》（*http://bit.ly/ 2HSsNBF*）
- 《使用 KNN 作为填充方法的研究》（*http://bit.ly/ 2HS9sAT*）

5.5 处理不均衡分类

问题描述

处理一个分类极度不均衡的目标向量。

解决方案

收集更多的数据。如果做不到，就改变评估模型的衡量标准。如果这也不起作用，可以考虑使用嵌入分类权重参数（如果有的话）的模型、下采样或上采样。评估标准会在后面的章节中介绍，现在我们先重点看一看分类权重参数、下采样和上采样。

为了方便演示，需要创建一些带有不均衡分类的数据。费雪鸢尾花数据集（Fisher's Iris dataset）包含 3 个均衡的分类，每个分类有 50 个观察值，都标注了花的种类：山鸢尾（Iris setosa）、维吉尼亚鸢尾（Iris virginia）和变色鸢尾（Iris versilocor）。为了得到不均衡的数据集，我们移除了 40 个山鸢尾的观察值，并将维吉尼亚鸢尾和变色鸢尾的数据合并。解决方案的目的是得到一个二元目标向量，表示一个观察值的分类是否为山鸢尾。最后显示数据中有 10 个观察值是山鸢尾（分类 0），有 100 个观察值不是山鸢尾（分类 1）：

```
# 加载库
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

# 加载鸢尾花数据
iris = load_iris()

# 创建特征矩阵
features = iris.data

# 创建目标向量
target = iris.target

# 移除前 40 个观察值
features = features[40:,:]
target = target[40:]

# 创建二元目标向量来标识观察值是否为类别 0
target = np.where((target == 0), 0, 1)

# 查看不均衡的目标向量
target
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

scikit-learn 的很多算法都提供了一个参数，可以在训练时对分类进行加权，以此抵消分类数据不均衡带来的影响。虽然之前没有提及，但是 RandomForestClassifier 是一个流行的分类算法，它也有一个 class_weight 参数。这个参数可以显式地指定想要的分类权重：

```
# 创建权重
weights = {0: .9, 1: 0.1}

# 创建带权重的随机森林分类器
RandomForestClassifier(class_weight=weights)

RandomForestClassifier(bootstrap=True, class_weight={0: 0.9, 1: 0.1},
            criterion='gini', max_depth=None, max_features='auto',
            max_leaf_nodes=None, min_impurity_decrease=0.0,
            min_impurity_split=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
            verbose=0, warm_start=False)
```

还可以传入参数 balanced，它会自动创建与分类的频数成反比的权重：

```
# 训练一个带均衡分类权重的随机森林分类器
RandomForestClassifier(class_weight="balanced")

RandomForestClassifier(bootstrap=True, class_weight='balanced',
            criterion='gini', max_depth=None, max_features='auto',
            max_leaf_nodes=None, min_impurity_decrease=0.0,
            min_impurity_split=None, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
            verbose=0, warm_start=False)
```

另外，对占多数的分类进行下采样和对占少数的分类进行上采样也是不错的方法。在下采样中，从占多数的分类（也就是拥有更多观察值的分类）无放回地随机取出观察值，创建一个观察值数量与占少数的分类相同的子集。例如，如果占少数的分类有 10 个观察值，那么就从占多数的分类中随机无放回地选取 10 个观察值，然后用这 20 个观察值作为数据集。下面是对于不均衡的山鸢尾花数据集所做的具体处理：

```
# 给每个分类的观察值打标签
i_class0 = np.where(target == 0)[0]
i_class1 = np.where(target == 1)[0]

# 确定每个分类的观察值数量
n_class0 = len(i_class0)
n_class1 = len(i_class1)

# 对于每个分类为 0 的观察值，从分类为 1 的数据中进行无放回的随机采样
i_class1_downsampled = np.random.choice(i_class1, size=n_class0,
replace=False)

# 将分类为 0 的目标向量和下采样的分类为 1 的目标向量连接起来
np.hstack((target[i_class0], target[i_class1_downsampled]))

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

# 将分类为 0 的特征矩阵和下采样的分类为 1 的特征矩阵连接起来
np.vstack((features[i_class0,:], features[i_class1_downsampled,:]))[0:5]

array([[ 5. ,  3.5,  1.3,  0.3],
       [ 4.5,  2.3,  1.3,  0.3],
       [ 4.4,  3.2,  1.3,  0.2],
       [ 5. ,  3.5,  1.6,  0.6],
       [ 5.1,  3.8,  1.9,  0.4]])
```

另一个选择是对占多数的分类进行上采样。在上采样中，针对占多数的分类，从占少数的分类中进行有放回的随机采样。在最后得到的结果中，占少数的分类和占多数的分类的观察值数量是相同的。上采样和下采样的实现方式相似，只是反转了一下：

```
# 对于每个分类为 1 的观察值，从分类为 0 的数据中进行有放回的随机采样
i_class0_upsampled = np.random.choice(i_class0, size=n_class1, replace=True)

# 将上采样得到的分类为 0 的目标向量和分类为 1 的目标向量连接起来
np.concatenate((target[i_class0_upsampled], target[i_class1]))

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

```
# 将上采样得到的分类为 0 的特征矩阵和分类为 1 的特征矩阵连接起来
np.vstack((features[i_class0_upsampled,:], features[i_class1,:]))[0:5]
```

```
array([[ 5. ,  3.5,  1.6,  0.6],
       [ 5. ,  3.5,  1.6,  0.6],
       [ 5. ,  3.3,  1.4,  0.2],
       [ 4.5,  2.3,  1.3,  0.3],
       [ 4.8,  3. ,  1.4,  0.3]])
```

讨论

在真实场景中，不均衡的分类到处可见，比如大多数访客都不会单击“购买”按钮，许多癌症也是比较罕见的。因此，处理不均衡分类就成为机器学习中的一个常见任务。

最好的解决方案是收集更多的观察值——尤其是占少数的分类的观察值。但是通常情况下，这是没法做到的，所以我们需要求助于其他手段。

次优的解决方案是选择更适用于评估不均衡数据的标准。准确率（accuracy）常常被作为评估模型性能的标准，但用准确率来评估不均衡的分类是不合适的。例如，如果样本中只有 0.5% 的人得了某种罕见的癌症，那么即便我们的模型预测没有人会得这个癌症，准确率也能达到 99.5%。很明显，这不是我们想要的结果。后面的章节会讨论更有效的评估标准，包括混淆矩阵、精确度、召回率、F1 值以及 ROC 曲线。

第 3 个解决方案是在一些分类器模型中使用分类权重参数，这样就能针对不均衡的分类来调整算法。幸运的是，很多 scikit-learn 的分类器都有 `class_weight` 参数，使用起来很方便。

第 4 和第 5 个解决方案是相关的：下采样和上采样。在下采样中，需要从占多数的分类中创建一个子集，其观察值数量与占少数的分类的观察值数量相等。在上采样中，采取有放回的方式对占少数的分类重复采样，以此创建与占多数的分类有相同数量观察值的数据集。到底是采用下采样还是上采样，需要根据场景做决定。通常情况下，应该同时尝试两种方法，看哪一种的效果更好。

第 6 章

处理文本

6.0 简介

非结构化文本数据（比如，一本书或一条推文的内容）是处理起来最有意思也是最难的特征之一。本章会介绍几种将文本转换成富含信息的特征的方法，但并未涵盖所有相关内容——针对这一类数据的处理已经形成一套完整的学科，所有的研究内容加起来写成书的话，能填满一个小型图书馆了。本章将介绍其中一些常用的技术，掌握了它们，对数据预处理工作是很有帮助的。

6.1 清洗文本

问题描述

对一些非结构化的文本数据进行基本的清洗。

解决方案

大部分基本的文本清洗操作用 Python 的常用字符串操作即可完成，其中 `strip`、`replace` 和 `split` 这三个操作用得最多：

```
# 创建文本
text_data = ["   Interrobang. By Aishwarya Henriette     ",
             "Parking And Going. By Karl Gautier",
             "    Today Is The night. By Jarek Prakash   "]

# 去除文本两端的空格
strip_whitespace = [string.strip() for string in text_data]
```

```
# 查看文本
strip_whitespace
['Interrobang. By Aishwarya Henriette',
 'Parking And Going. By Karl Gautier',
 'Today Is The night. By Jarek Prakash']

# 删除句点
remove_periods = [string.replace(".", "") for string in strip_whitespace]

# 查看文本
remove_periods
['Interrobang By Aishwarya Henriette',
 'Parking And Going By Karl Gautier',
 'Today Is The night By Jarek Prakash']
```

也可以创建并应用自定义的转换函数：

```
# 创建函数
def capitalizer(string: str) -> str:
    return string.upper()

# 应用函数
[capitalizer(string) for string in remove_periods]

['INTERROBANG BY AISHWARYA HENRIETTE',
 'PARKING AND GOING BY KARL GAUTIER',
 'TODAY IS THE NIGHT BY JAREK PRAKASH']
```

最后，还可以使用正则表达式来做一些复杂的字符串操作：

```
# 加载库
import re

# 创建函数
def replace_letters_with_X(string: str) -> str:
    return re.sub(r"[a-zA-Z]", "X", string)

# 应用函数
[replace_letters_with_X(string) for string in remove_periods]

['XXXXXXXXXXX XX XXXXXXXXX XXXXXXXXX',
 'XXXXXXX XXX XXXXX XX XXXX XXXXXXX',
 'XXXXX XX XXX XXXXX XX XXXXX XXXXXXX']
```

讨论

大多数文本数据在被用于生成特征之前都需要进行清洗。Python 的标准字符串操作能完成大部分基本的文本清洗操作。在现实场景中，我们通常需要自定义清洗函数（比如，`capitalizer`）完成一些组合的清洗任务，再将该函数应用于文本数据。

延伸阅读

- 《Python 正则表达式的初学者教程》（*http://bit.ly/2HTGZuu*）

6.2 解析并清洗 HTML

问题描述

提取 HTML 文件中的文本数据，而且仅提取某段文本。

解决方案

Beautiful Soup 是一个可以从 HTML 或 XML 文件中提取数据的 Python 库，提供了很多选项。我们用它对 HTML 进行解析并提取数据：

```
# 加载库
from bs4 import BeautifulSoup

# 创建一些 HTML 代码
html = """
       <div class='full_name'><span style='font-weight:bold'>
       Masego Azra"
       """

# 解析 HTML
soup = BeautifulSoup(html, "lxml")

# 查找 class 是 "full_name" 的 div 标签，并查看文本
soup.find("div", { "class" : "full_name" }).text

'Masego Azra'
```

讨论

尽管 Beautiful Soup 这个名字很奇怪，但它在处理 HTML 方面的能力还是很强大的。通常，Beautiful Soup 是用于抓取在线网页的，不过它也可以很轻松地提取嵌在 HTML 中的文本。

本书不会对 Beautiful Soup 操作进行全面介绍，这超出了本书的范围。从上述解决方案可以看出，用它解析 HTML 代码并提取想要的数据非常简单。

延伸阅读

- Beautiful Soup（*http://bit.ly/2pwZcYs*）

6.3 移除标点

问题描述

移除文本数据的特征中的标点。

解决方案

定义一个函数，将一个标点字典作为参数传入 `translate`：

```
# 加载库
import unicodedata
import sys

# 创建文本
text_data = ['Hi!!!! I. Love. This. Song....',
             '10000% Agree!!!! #LoveIT',
             'Right?!?!']

# 创建一个标点字典
punctuation = dict.fromkeys(i for i in range(sys.maxunicode)
                            if unicodedata.category(chr(i)).startswith('P'))

# 移除每个字符串中的标点
[string.translate(punctuation) for string in text_data]

['Hi I Love This Song', '10000 Agree LoveIT', 'Right']
```

讨论

`translate` 因其非凡的性能成为 Python 中非常流行的函数。上述解决方案先创建一个 `punctuation` 字典，将 Unicode 中的标点字符作为 key，`None` 作为 value，然后将字符串中所有在 `punctuation` 字典中出现过的字符（即标点）转换成 `None`，高效地移除它们。当然还有更易于理解的移除标点的方案，这个方案有点像黑客解决方案，但是它远快于其他的方案。

标点也包含信息（对比一下“是？”和“是！”），认识到这个事实是很重要的。在创建特征时难免要删除标点，但是如果这个标点很重要，还是要考虑将其保留。

6.4 文本分词

问题描述

将文本分离成独立的单词。

解决方案

Python 的自然语言工具集（Natural Language Toolkit，NLTK）在处理文本方面有很多功能强大的操作，其中包括文本分词（word tokenizing）：

```
# 加载库
from nltk.tokenize import word_tokenize

# 创建文本
string = "The science of today is the technology of tomorrow"

# 分词
word_tokenize(string)

['The', 'science', 'of', 'today', 'is', 'the', 'technology', 'of',
'tomorrow']
```

也可以将文本切分为句子：

```
# 加载库
from nltk.tokenize import sent_tokenize

# 创建文本
string = "The science of today is the technology of tomorrow. Tomorrow is
today."

# 切分成句子
sent_tokenize(string)

['The science of today is the technology of tomorrow.', 'Tomorrow is today.']
```

讨论

文本分词，尤其是将文本分为独立的单词是在清洗文本数据之后的常见任务，因为它是将文本转换成能用于构建有效特征的数据的第一步。

6.5 删除停止词（stop word）

问题描述

给定一组分好词的文本数据，删除其中非常常见但包含的信息又很少的单词（例如，a、is、of、on）。

解决方案

使用 NLTK 的 `stopwords`：

```
# 加载库
from nltk.corpus import stopwords

# 如果是第一次使用停止词集，则需要先下载
# import nltk
# nltk.download('stopwords')

# 创建单词序列
tokenized_words = ['i',
                   'am',
                   'going',
                   'to',
                   'go',
                   'to',
                   'the',
                   'store',
                   'and',
                   'park']

# 加载停止词
stop_words = stopwords.words('english')

# 删除停止词
[word for word in tokenized_words if word not in stop_words]

['going', 'go', 'store', 'park']
```

讨论

尽管“停止词”可以指代所有需要在数据预处理阶段删除的单词，但是这个术语常常用来指代那些特别常见而包含的信息又很少的单词。NLTK 有一个常见停止词列表，可用来查找并删除单词序列中的停止词：

```
# 查看停止词
stop_words[:5]

['i', 'me', 'my', 'myself', 'we']
```

注意，NLTK 的 stopwords 假设所有的单词都是小写形式的。

6.6 提取词干

问题描述

将一个单词序列中的单词转换成它们的词干。

解决方案

使用 NLTK 的 PorterStemmer：

```
# 加载库
from nltk.stem.porter import PorterStemmer

# 创建单词序列
tokenized_words = ['i', 'am', 'humbled', 'by', 'this', 'traditional',
'meeting']

# 创建词干转换器
porter = PorterStemmer()

# 应用词干转换器
[porter.stem(word) for word in tokenized_words]

['i', 'am', 'humbl', 'by', 'thi', 'tradit', 'meet']
```

讨论

词干提取（stemming）能识别出一个单词的词缀并将其删除（例如，动名词中的“ing”后缀），同时保留其词根的意思，以此得到这个词的词干。例如，“tradition”和“traditional”的词干都是“tradit”，表明虽然它们是不同的单词，但基本意思是相同的。提取文本的词干后，单词的可读性会变差，不过也更接近它的基本意思，因此更适合用来做比较。

NLTK 的 PorterStemmer 实现了被广泛使用的波特词干（Porter stemming）算法，移除或替换单词中常用的后缀来生成词干。

延伸阅读

- 波特词干算法（*http://bit.ly/2FB5ZZb*）

6.7 标注词性

问题描述

为文本数据中的每个单词或字符标注词性。

解决方案

使用 NLTK 预训练的词性标注器：

```
# 加载库
from nltk import pos_tag
from nltk import word_tokenize

# 创建文本
text_data = "Chris loved outdoor running"

# 使用预训练的词性标注器
text_tagged = pos_tag(word_tokenize(text_data))

# 查看词性
text_tagged
```

```
[('Chris', 'NNP'), ('loved', 'VBD'), ('outdoor', 'RP'), ('running', 'VBG')]
```

上述代码输出一个包含单词和其词性的元组列表。NLTK 使用 Penn Treebank 的词性标签进行标注。这里给出一部分 Penn Treebank 标签的例子，如下表所示。

标签	词性
NNP	单数专有名词
NN	单数或复数的名词
RB	副词
VBD	过去式的动词
VBG	动名词或动词的现在分词形式
JJ	形容词
PRP	人称代词

一旦为文本添加了词性标签，就能依据标签找到特定词性的单词了。例如，找到所有的名词：

```
# 过滤单词
[word for word, tag in text_tagged if tag in ['NN','NNS','NNP','NNPS'] ]

['Chris']
```

现实中更可能遇到的情况是，有一份数据，其中每个观察值都包含一条推文，我们想将这些句子转换成用词性表示的特征（例如，如果是专有名词，特征值就是 1，否则为 0）：

```
# 创建文本
tweets = ["I am eating a burrito for breakfast",
          "Political science is an amazing field",
          "San Francisco is an awesome city"]

# 创建列表
tagged_tweets = []

# 为每条推文中的每个单词加标签
for tweet in tweets:
    tweet_tag = nltk.pos_tag(word_tokenize(tweet))
    tagged_tweets.append([tag for word, tag in tweet_tag])

# 使用 one-hot 编码将标签转换成特征
one_hot_multi = MultiLabelBinarizer()
one_hot_multi.fit_transform(tagged_tweets)

array([[1, 1, 0, 1, 0, 1, 1, 1, 0],
       [1, 0, 1, 1, 0, 0, 0, 0, 1],
       [1, 0, 1, 1, 1, 0, 0, 0, 1]])
```

使用 classes_ 就能发现每个特征都是一个词性标签：

```
# 查看特征名
one_hot_multi.classes_

array(['DT', 'IN', 'JJ', 'NN', 'NNP', 'PRP', 'VBG', 'VBP', 'VBZ'],
dtype=object)
```

讨论

如果文本是英文的，而且不是关于某个特定主题的（例如医学），那么标注词性最简单的做法就是使用 NLTK 预训练的词性标注器。但是，如果 `pos_tag` 不是很准确，NLTK 也提供了自行训练标注器的机制。训练标注器的主要缺点就是需要一个很大的文本语料库，并且其中每个单词的词性标签都是已知的。构建这样的带标签的语料库显然是一件很烦琐的工作，所以只能作为最后没有办法的办法。

说了这么多，如果你想要用带标签的语料库来训练标注器，可以参照下面例子的做法。我们使用的语料库是 Brown Copus（布朗语料库），它是最流行的带标签的文本语料库之一。我们使用的是回退 *n* 元模型（back *n*-gram）标注器，*n* 表示在预测一个单词的词性时需要考虑多少个前面的单词（其实是 *n*–1 个）。首先，考虑这个单词的前两个单词（使用 `TrigramTagger`），如果它前面没有两个单词，则“回退”到只考虑它前面的一个单词，这时候使用 `BigramTagger`。如果这个单词前面一个单词都没有，就通过 `UnigramTagger` 只考虑这个单词本身。为了检测标注器的准确率，我们将文本数据分成两个部分，一部分用来训练标注器，另一部分用来测试标注器的效果：

```
# 加载库
from nltk.corpus import brown
from nltk.tag import UnigramTagger
from nltk.tag import BigramTagger
from nltk.tag import TrigramTagger

# 从布朗语料库中获取文本数据，切分成句子
sentences = brown.tagged_sents(categories='news')

# 将 4000 个句子用作训练，623 个句子用作测试
train = sentences[:4000]
test = sentences[4000:]

# 创建回退标注器
unigram = UnigramTagger(train)
bigram = BigramTagger(train, backoff=unigram)
trigram = TrigramTagger(train, backoff=bigram)

# 查看准确率
trigram.evaluate(test)

0.8179229731754832
```

延伸阅读

- Penn Treebank（*http://bit.ly/2HROPo5*）
- 布朗语料库（*http://bit.ly/2FzgMTx*）

6.8 将文本编码成词袋（Bag of Words）

问题描述

创建一组特征来表示观察值文本中包含的特定单词的数量。

解决方案

使用 scikit-learn 的 `CountVectorizer`：

```
# 加载库
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

# 创建文本
text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])

# 创建一个词袋特征矩阵
count = CountVectorizer()
bag_of_words = count.fit_transform(text_data)

# 查看特征矩阵
bag_of_words

<3x8 sparse matrix of type ''
    with 8 stored elements in Compressed Sparse Row format>
```

上述代码的输出是一个稀疏矩阵。当文本量很大时，使用这样的存储方式是有必要的。但是在我们的玩具例子中，可以使用 `toarray` 查看每个观察值的词频统计矩阵：

```
bag_of_words.toarray()

array([[0, 0, 0, 2, 0, 0, 1, 0],
       [0, 1, 0, 0, 0, 1, 0, 1],
       [1, 0, 1, 0, 1, 0, 0, 0]], dtype=int64)
```

使用 `get_feature_names` 方法能查看每个特征所对应的单词：

```
# 查看特征名
count.get_feature_names()
```

```
['beats', 'best', 'both', 'brazil', 'germany', 'is', 'love', 'sweden']
```

这样的输出可能有点令人困惑。为了说明特征矩阵到底是什么样子的，下面以单词为列名（每一行就是一个观察值）给出了具体示例。

beats	best	both	brazil	germany	is	love	sweden
0	0	0	2	0	0	1	0
0	1	0	0	0	1	0	1
1	0	1	0	1	0	0	0

讨论

将文本转换成特征的最常用的方法之一就是使用词袋模型（bag-of-words model）。词袋模型为文本数据中的每一个单词都输出一个特征，每个特征都包含该单词在观察值中出现的次数。例如，在我们的解决方案中，因为 brazil 在句子“I love Brazil. Brazil!”中出现了两次，所以特征“brazil”的值是 2。

上述解决方案中的文本数据有意选得很小，但是在现实场景中，一个文本数据的观察值可能是整本书的内容！因为词袋模型为数据中的每个单词都创建了一个特征，所以生成的矩阵会包含成千上万个特征，也就意味着这个矩阵有时需要占用很大的内存。但幸运的是，我们可以利用词袋特征矩阵的特性来减少需要存储的数据量。

大多数观察值并不会包含太多单词，因此词袋特征矩阵里会有很多值为 0。这样的矩阵被称为“稀疏”矩阵。可以只存储非零值并假设所有其他值都是 0，而不是将矩阵中的所有值都存储下来，这样当特征矩阵很大时就能节省很多内存空间。`CountVectorizer` 有一个很好的特性，就是默认以稀疏矩阵作为输出。

`CountVectorizer` 还有很多有用的参数，它们能让创建词袋特征矩阵变得很简单。首先，尽管默认情况下每个特征是一个单词，但也并非必须如此。我们可以将特征设置成两个词（称为二元模型，2-gram）或三个词（三元模型，3-gram）的组合。`ngram_range` 为我们的 *n* 元模型设置最大元和最小元，例如，`(2,3)` 会返回所有的二元模型和三元模型。其次，使用 `stop_words` 也可以很方便地移除信息量比较少的单词，停止词列表可以是函数自带的也可以是自定义的。最后，使用 `vocabulary` 可以将观察值限定为仅在特定的单词表中出现过的单词或短语。例如，创建一个只包含国家名字的词袋特征矩阵 ：

```
# 用特定参数创建特征矩阵
count_2gram = CountVectorizer(ngram_range=(1,2),
                              stop_words="english",
```

```
                                vocabulary=['brazil'])
bag = count_2gram.fit_transform(text_data)

# 查看特征矩阵
bag.toarray()

array([[2],
       [0],
       [0]])

# 查看一元模型和二元模型
count_2gram.vocabulary_

{'brazil': 0}
```

延伸阅读

- *n*-gram（*https://en.wikipedia.org/wiki/N-gram*）
- Bag of Words Meets Bags of Popcorn [1]（*http://bit.ly/2HRba5v*）

6.9 按单词的重要性加权

问题描述

对词袋模型中的词按照其在观察值中的重要程度进行加权。

解决方案

使用 TF-IDF（term frequency-inverse document frequency ）将一个词在某个文档（推文、影评、演讲稿等）中的出现次数和这个词在所有文档中的出现次数进行对比。用 scikit-learn 的 `TfidfVectorizer` 能很方便地做这个对比：

```
# 加载库
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

# 创建文本
text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])
```

1 编者注：这是一个 Kaggle 案例，讲的是 word2vec 在情感分析上的应用。

```
# 创建 TF-IDF 特征矩阵
tfidf = TfidfVectorizer()
feature_matrix = tfidf.fit_transform(text_data)

# 查看 TF-IDF 特征矩阵
feature_matrix

<3x8 sparse matrix of type ''
    with 8 stored elements in Compressed Sparse Row format>
```

和 6.8 节一样，TfidfVectorizer 的输出也是一个稀疏矩阵。不过，如果想看稠密矩阵形式的输出，可以使用 .toarray：

```
# 查看 TF-IDF 特征矩阵的稠密矩阵形式
feature_matrix.toarray()

array([[0.        , 0.        , 0.        , 0.89442719, 0.        ,
        0.        , 0.4472136 , 0.        ],
       [0.        , 0.57735027, 0.        , 0.        , 0.        ,
        0.57735027, 0.4472136 , 0.57735027],
       [0.57735027, 0.        , 0.57735027, 0.        , 0.57735027,
        0.        , 0.        , 0.        ]])
```

vocabulary_ 可以查看特征的单词表：

```
# 查看特征的名字
tfidf.vocabulary_

{'beats': 0,
 'best': 1,
 'both': 2,
 'brazil': 3,
 'germany': 4,
 'is': 5,
 'love': 6,
 'sweden': 7}
```

讨论

一个词在文档中出现的次数越多，它对这个文档就越重要。例如，如果单词 *economy* 在某篇文档中频繁出现，就表示这个文档可能是关于经济的。我们把单词在文档中的出现次数称为词频（*term frequency*，*tf*）。

相反，如果一个单词在很多文档中都出现了，那么它对于单个文档的重要性就没有那么高了。例如，如果每个文档中的文本数据都包含单词 after，那么它很可能不是一个

重要的词。一个单词出现在了多少文档中，我们把这个数值称为文档频率（*document frequency*，*df*）。

将这两个统计值组合起来，我们就能为每个单词确定一个分数，来表示它在某个文档中的重要程度了。具体地说，就是将 *tf* 和 *idf*（*inverse of document frequency*，逆向文档频率）相乘：

$$\text{tf-idf}(t, d) = tf(t, d) \cdot idf(t)$$

这里，*t* 表示单词，*d* 表示文档。计算 *tf* 和 *idf* 的方法有很多。在 scikit-learn 中，就是单词在文档中出现的次数，*idf* 是这样计算出来的：

$$\text{idf}(t) = log\frac{1 + n_d}{1 + \text{df}(d, t)} + 1$$

n_d 是文档的数量，*df*(*d*,*t*) 是单词 *t* 的文档频率（也就是单词在多少份文档中出现过）。

默认情况下，scikit-learn 会使用欧氏范数（L2 范数）将 TF-IDF 向量归一化。得到的结果值越大，这个单词对一个文档来说就越重要。

延伸阅读

- scikit-learn 文档：TF-IDF 权重（*http://bit.ly/2HT2wmW*）

第 7 章

处理日期和时间

7.0 简介

对数据做预处理时常常会遇到日期和时间（datetime）类型的数据。比如某次销售发生的时间或者某次公民健康数据统计的年份。本章会介绍处理时间序列数据的一系列策略，包括处理时区和创建滞后的时间序列特征。我们会重点关注 pandas 中的时间序列工具，它把很多其他库的功能集中到了一起。

7.1 把字符串转换成日期

问题描述

把一个代表日期和时间的字符串向量转换成时间序列数据。

解决方案

使用 pandas 的 to_datetime 函数，并通过 format 参数指定字符串的日期和（或）时间格式：

```
# 加载库
import numpy as np
import pandas as pd

# 创建字符串
date_strings = np.array(['03-04-2005 11:35 PM',
                         '23-05-2010 12:01 AM',
                         '04-09-2009 09:09 PM'])
```

```
# 转换成 datetime 类型的数据
[pd.to_datetime(date, format='%d-%m-%Y %I:%M %p') for date in date_strings]

[Timestamp('2005-04-03 23:35:00'),
 Timestamp('2010-05-23 00:01:00'),
 Timestamp('2009-09-04 21:09:00')]
```

还可以增加一个 errors 参数来处理错误：

```
# 转换成 datetime 类型的数据
[pd.to_datetime(date, format="%d-%m-%Y %I:%M %p", errors="coerce")
for date in date_strings]

[Timestamp('2005-04-03 23:35:00'),
 Timestamp('2010-05-23 00:01:00'),
 Timestamp('2009-09-04 21:09:00')]
```

如果传入 errors="coerce"，当转换出现错误时不会抛出异常（默认行为），但是会将导致这个错误的值设置成 NaT（也就是缺失值）。

讨论

当日期和时间以字符串形式出现时，需要将它们转换成 Python 能理解的数据类型。虽然 Python 中有许多工具能将字符串转换成 datetime 类型，但在后面的章节中我们都会使用 pandas 的 to_datetime 来执行转换。用字符串来表示日期和时间有一个缺点，即如果数据源不同，字符串的格式也会相去甚远。例如，一个日期向量可能将 2015 年 3 月 23 日表示成“03-23-15”，也可能表示成“3|23|2015”。使用 format 参数可以指定字符串的具体格式。下表列出了常用的日期和时间的格式化代码。

代码	描述	例子
%Y	完整的年份	2001
%m	月，首位空缺时需用 0 填充	04
%d	日，首位空缺时需用 0 填充	09
%I	小时，首位空缺时需用 0 填充	02
%p	AM（上午）或 PM（下午）	AM
%M	分，首位空缺时需用 0 填充	05
%S	秒，首位空缺时需用 0 填充	09

延伸阅读

- Python 中日期和时间格式化代码的完整列表（*http://strftime.org*）

7.2 处理时区

问题描述

为一组时间序列数据添加或改变时区。

解决方案

如果没有特别指定，pandas 的对象都是没有时区的，不过可以在创建对象时通过 `tz` 参数来指定时区：

```
# 加载库
import pandas as pd

# 创建 datetime
pd.Timestamp('2017-05-01 06:00:00', tz='Europe/London')
Timestamp('2017-05-01 06:00:00+0100', tz='Europe/London')
```

可以用 `tz_localize` 为之前创建的 `datetime` 对象添加时区信息：

```
# 创建 datetime
date = pd.Timestamp('2017-05-01 06:00:00')

# 设置时区
date_in_london = date.tz_localize('Europe/London')

# 查看 datetime
date_in_london

Timestamp('2017-05-01 06:00:00+0100', tz='Europe/London')
```

还可以转换时区：

```
# 改变时区
date_in_london.tz_convert('Africa/Abidjan')

Timestamp('2017-05-01 05:00:00+0000', tz='Africa/Abidjan')
```

最后，pandas 的 Series 对象能对每一个元素应用 `tz_localize` 和 `tz_convert`：

```
# 创建 3 个日期
dates = pd.Series(pd.date_range('2/2/2002', periods=3, freq='M'))

# 设置时区
dates.dt.tz_localize('Africa/Abidjan')
```

```
0   2002-02-28 00:00:00+00:00
1   2002-03-31 00:00:00+00:00
2   2002-04-30 00:00:00+00:00
dtype: datetime64[ns, Africa/Abidjan]
```

讨论

pandas 支持两种表示时区的字符串，但是我建议使用 pytz 库的字符串。导入 `all_timezones` 库就能看到所有代表时区的字符串：

```
# 加载库
from pytz import all_timezones

# 查看两个时区
all_timezones[0:2]
```

```
['Africa/Abidjan', 'Africa/Accra']
```

7.3 选择日期和时间

问题描述

从一组日期向量中选择一个或多个日期值。

解决方案

使用两个布尔条件句分别设置开始日期和结束日期：

```
# 加载库
import pandas as pd

# 创建数据帧
dataframe = pd.DataFrame()

# 创建 datetime
dataframe['date'] = pd.date_range('1/1/2001', periods=100000, freq='H')

# 筛选出两个日期之间的观察值
dataframe[(dataframe['date'] > '2002-1-1 01:00:00') &
          (dataframe['date'] <= '2002-1-1 04:00:00')]
```

	date
8762	2002-01-01 02:00:00
8763	2002-01-01 03:00:00
8764	2002-01-01 04:00:00

或者，将 date（日期）这一列设为数据帧的索引列，然后用 loc 进行筛选：

```
# 设置索引
dataframe = dataframe.set_index(dataframe['date'])

# 选择两个日期之间的观察值
dataframe.loc['2002-1-1 01:00:00':'2002-1-1 04:00:00']
```

	date
2002-01-01 01:00:00	2002-01-01 01:00:00
2002-01-01 02:00:00	2002-01-01 02:00:00
2002-01-01 03:00:00	2002-01-01 03:00:00
2002-01-01 04:00:00	2002-01-01 04:00:00

讨论

你需要根据场景来决定，是用布尔条件句还是用索引来筛选日期值。如果你想执行复杂的时间序列操作，那么将 date 这一列设置为数据帧的索引是值得的；但是如果你只是想做一些简单的数据整理工作，那么使用布尔条件句会更简单一些。

7.4 将日期数据切分成多个特征

问题描述

用一列日期和时间数据来创建年、月、日、时、分的特征。

解决方案

使用 pandas 函数 Series.dt 的时间属性：

```
# 加载库
import pandas as pd

# 创建数据帧
dataframe = pd.DataFrame()

# 创建 5 个日期
dataframe['date'] = pd.date_range('1/1/2001', periods=150, freq='W')
```

```
# 创建年、月、日、时和分的特征
dataframe['year'] = dataframe['date'].dt.year
dataframe['month'] = dataframe['date'].dt.month
dataframe['day'] = dataframe['date'].dt.day
dataframe['hour'] = dataframe['date'].dt.hour
dataframe['minute'] = dataframe['date'].dt.minute

# 查看3行
dataframe.head(3)
```

	date	year	month	day	hour	minute
0	2001-01-07	2001	1	7	0	0
1	2001-01-14	2001	1	14	0	0
2	2001-01-21	2001	1	21	0	0

讨论

有时候将 date 这一列划分成多个部分是很有用的。比如，你可能需要用一个特征来表示观察值的年份；或者你仅希望考虑观察值的月份以便对它们进行比较，而不管其年份是多少。

7.5 计算两个日期之间的时间差

问题描述

为每个观察值计算两个日期特征之间的时间差。

解决方案

使用 pandas 对两个日期特征做减法：

```
# 加载库
import pandas as pd

# 创建数据帧
dataframe = pd.DataFrame()

# 创建两个 datetime 特征
dataframe['Arrived'] = [pd.Timestamp('01-01-2017'),pd.Timestamp('01-04-2017')]
dataframe['Left'] = [pd.Timestamp('01-01-2017'), pd.Timestamp('01-06-2017')]
```

```
# 计算两个特征之间的时间间隔
dataframe['Left'] - dataframe['Arrived']

0   0 days
1   2 days
dtype: timedelta64[ns]
```

我们经常会遇到这样的情况，即需要移除输出中的 days 字符串，仅留下数值：

```
# 计算两个特征之间的时间间隔
pd.Series(delta.days for delta in (dataframe['Left'] - dataframe['Arrived']))

0   0
1   2
dtype: int64
```

讨论

有时我们想要的特征是两个时间点之间的时间间隔（delta）。例如，我们知道一位客户在酒店的入住和退房时间，而我们想要的特征是他住了几天。pandas 的 `TimeDelta` 数据类型可以让这件事情变得很简单。

延伸阅读

- pandas 文档：Time Deltas（*http://bit.ly/2HOS3sJ*）

7.6 对一周内的各天进行编码

问题描述

求一个日期向量中的每一个日期是星期几。

解决方案

使用 pandas 中 `Series.dt` 的 `weekday_name` 属性：

```
# 加载库
import pandas as pd

# 创建日期
dates = pd.Series(pd.date_range("2/2/2002", periods=3, freq="M"))

# 查看星期几
```

```
dates.dt.weekday_name

0    Thursday
1      Sunday
2     Tuesday
dtype: object
```

如果希望输出是一个数值，以便其成为一个更有用的机器学习特征，则可以使用weekday属性。它用一个整数来表示一个星期中的星期几（星期一是0）：

```
# 查看星期几
dates.dt.weekday

0   3
1   6
2   1
dtype: int64
```

讨论

在一些场景下，知道某一天是星期几会很有用，比如要比较过去3年里每个星期日的销售总额。而创建一个特征向量，其中包含星期几这样的信息，在pandas中是很容易做到的。

延伸阅读

- pandas的Series对象中和日期有关的属性（*http://bit.ly/2HShhq1*）

7.7 创建一个滞后的特征

问题描述

创建一个滞后 *n* 个时间段的特征。

解决方案

使用pandas的`shift`：

```
# 加载库
import pandas as pd

# 创建数据帧
dataframe = pd.DataFrame()
```

```
# 创建数据
dataframe["dates"] = pd.date_range("1/1/2001", periods=5, freq="D")
dataframe["stock_price"] = [1.1,2.2,3.3,4.4,5.5]

# 让值滞后一行
dataframe["previous_days_stock_price"] = dataframe["stock_price"].shift(1)

# 查看数据帧
dataframe
```

	dates	stock_price	previous_days_stock_price
0	2001-01-01	1.1	NaN
1	2001-01-02	2.2	1.1
2	2001-01-03	3.3	2.2
3	2001-01-04	4.4	3.3
4	2001-01-05	5.5	4.4

讨论

通常情况下，数据之间的时间间隔是固定的（例如，一天、一小时、三小时），而我们又常常对使用历史数据做预测（这被称为*滞后一个特征*）感兴趣。例如，使用前一天的股票价格来预测今天的价格。有了 pandas，我们可以使用 `shift` 让数值滞后一行，创建一个包含历史数值的新特征。

在解决方案中，因为没有前一天的 `stock_price` 值，所以 `previous_days_stock_price` 的第一行是一个缺失值。

7.8 使用滚动时间窗口

问题描述

计算一个时间序列数据针对某个滚动时间的统计量。

解决方案

```
# 加载库
import pandas as pd

# 创建 datetime
time_index = pd.date_range("01/01/2010", periods=5, freq="M")
```

```
# 创建数据帧，设置索引
dataframe = pd.DataFrame(index=time_index)

# 创建特征
dataframe["Stock_Price"] = [1,2,3,4,5]

# 计算滚动平均值
dataframe.rolling(window=2).mean()
```

	Stock_Price
2010-01-31	NaN
2010-02-28	1.5
2010-03-31	2.5
2010-04-30	3.5
2010-05-31	4.5

讨论

滚动（有时候也称为移动）时间窗口的概念很简单，但刚接触它的时候理解起来可能会有点困难。假设我们对股票价格的观察值是以月为单位的。如果拥有一个确定月份数量的时间窗口，并且在所有的观察值中移动这个窗口，那么计算时间窗口中所有观察值的统计量是很有价值的。

例如，如果有一个宽度为 3 个月的时间窗口，要求滚动平均值，可以这样计算：

1. 求 1 月、2 月和 3 月的平均值。
2. 求 2 月、3 月和 4 月的平均值。
3. 求 3 月、4 月和 5 月的平均值。
4. 依此类推。

也就是说，我们用一个宽度为 3 个月的时间窗口“走”过了所有的观察值，每走一步，都会计算这个窗口内所有观察值的平均值。

pandas 的 `rolling` 可以通过 `window` 参数指定窗口的大小，同时还能很方便地计算一些常见的统计值，包括最大值（`max()`）、平均值（`mean()`）、计数（`count()`）和滚动相关性（`corr()`）。

滚动平均值常被用于对时间序列数据做平滑处理，因为使用整个时间窗口的平均值能削弱短期波动的影响。

延伸阅读

- pandas 文档：滚动窗口（*http://bit.ly/2HTIicA*）
- 《什么是移动平均或平滑技术》（*http://bit.ly/2HRRHBn*）

7.9 处理时间序列中的缺失值

问题描述

处理时间序列数据中的缺失值。

解决方案

除了之前讨论过的针对缺失数据的处理策略，针对时间序列数据，还可以使用插值法来填充由缺失值造成的数据缺口：

```
# 加载库
import pandas as pd
import numpy as np

# 创建日期
time_index = pd.date_range("01/01/2010", periods=5, freq="M")

# 创建数据帧，设置索引
dataframe = pd.DataFrame(index=time_index)

# 创建带缺失数据的特征
dataframe["Sales"] = [1.0,2.0,np.nan,np.nan,5.0]

# 对缺失数据进行插值
dataframe.interpolate()
```

	Sales
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	3.0
2010-04-30	4.0
2010-05-31	5.0

或者，可以用前面的值来替换缺失值（也就是向前填充）：

```
# 向前填充
dataframe.ffill()
```

	Sales
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	2.0
2010-04-30	2.0
2010-05-31	5.0

还可以用后面的值来替换缺失值（也就是向后填充）：

```
# 向后填充
dataframe.bfill()
```

	Sales
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	5.0
2010-04-30	5.0
2010-05-31	5.0

讨论

插值法是一种填充由缺失值造成的数据缺口的技术，实际上就是根据缺口附近的已知数据来画一条直线或者曲线，然后利用这条直线或曲线预测合理的值。当时间间隔确定，数据不会产生剧烈波动且缺失值缺口比较小的时候，插值法尤其有用。例如，上述解决方案中，缺失值缺口附近的值是 `2.0` 和 `5.0`。从 `2.0` 到 `5.0` 画一条线，根据这条线预测出两个缺失值比较合理的值是 `3.0` 和 `4.0`。

如果我们认为两个已知点之间的线是非线性的，可以使用 `interpolate` 的 `method` 参数来指定插值的方式：

```
# 对缺失值进行插值
dataframe.interpolate(method="quadratic")
```

	Sales
2010-01-31	1.000000
2010-02-28	2.000000
2010-03-31	3.059808
2010-04-30	4.038069
2010-05-31	5.000000

最后，缺失值的缺口可能非常大，我们不想对整个缺口进行插值。在这种情况下，可以使用 `limit` 来限制插值的数量，并用 `limit_direction` 来设置是从最后一个已知值进行向前插值，还是进行反向的操作。

```
 # 对缺失值进行插值
dataframe.interpolate(limit=1, limit_direction="forward")
```

	Sales
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	3.0
2010-04-30	NaN
2010-05-31	5.0

向后填充和向前填充是简单的插值法，它们都是画一条经过已知值的线，并利用这条线来填充缺失值。向后填充和向前填充还有一个小优点，即不需要知道缺失值两侧的值。

第 8 章

图像处理

8.0 简介

图像分类是机器学习中最令人兴奋的领域之一。从图像中识别出图案和物体的能力是机器学习算法的一种非常强大的能力。然而，在将机器学习算法应用于图像之前，通常需要先将原始图像转换为算法可用的特征。

我们将使用开源计算机视觉库（OpenCV）来处理图像。虽然有许多优秀的图像处理库，但 OpenCV 最流行，其文档也是最详细的。使用 OpenCV 时最大的障碍之一就是安装。然而幸运的是，如果使用 Python 3（在本书出版时 OpenCV 还不能与 Python 3.6 以上的版本一起使用），就可以使用 Anaconda 的包管理器工具 conda 来安装 OpenCV，只需要在终端中执行下面这一行命令即可：

```
conda install --channel    https://conda.anaconda.org/menpoopencv3
```

之后，可以打开一个记事本，导入 OpenCV 并检查版本号（3.1.0），以查看 OpenCV 的安装情况：

```
import cv2

cv2.__version__
```

如果使用 conda 安装 OpenCV 时遇到问题，在网络上可以找到很多教程。

本章将使用一组示例图像，你可以从 GitHub 下载这些数据（*https://github.com/chrisalbon/simulated_datasets*）。

8.1 加载图像

问题描述

为了对图像进行预处理，需要将其载入程序中。

解决方案

使用 OpenCV 的 imread 方法：

```
# 加载库
    import cv2
    import numpy as np
    from matplotlib import pyplot as plt


    # 把图像导入成灰度图
    image = cv2.imread("images/plane.jpg", cv2.IMREAD_GRAYSCALE)
```

如果想查看图像，可以使用 Python 的绘图库 Matplotlib：

```
# 显示图像
    plt.imshow(image, cmap="gray"), plt.axis("off")
    plt.show()
```

讨论

从本质上说，图像也是数据，而且调用 imread 方法时，程序会将图像数据转换为我们非常熟悉的数据类型—— NumPy 数组：

```
# 显示数据类型
    type(image)

    numpy.ndarray
```

我们已经把这张图像转换成一个矩阵，矩阵中的元素与原图像中的像素点一一对应。用下面的命令可以查看矩阵的值：

```
# 显示图像数据
image

array([[140, 136, 146, ..., 132, 139, 134],
       [144, 136, 149, ..., 142, 124, 126],
       [152, 139, 144, ..., 121, 127, 134],
       ...,
       [156, 146, 144, ..., 157, 154, 151],
       [146, 150, 147, ..., 156, 158, 157],
       [143, 138, 147, ..., 156, 157, 157]], dtype=uint8)
```

图像的分辨率是 3600 × 2270，和矩阵的维度完全相同：

```
# 显示矩阵维度
image.shape

(2270, 3600)
```

矩阵中的每个元素实际代表什么呢？对于灰度图像，矩阵元素的值代表对应像素的灰度值。灰度值在 0（黑色）到 255（白色）之间变化。例如，本节“解决方案”部分的图像中左上角像素点的灰度值为 140：

```
# 显示第一个像素点的像素值
image[0,0]


140
```

在下面的矩阵中，每个元素包含三个值，分别对应蓝色、绿色和红色分量（BGR）：

```
# 以彩色模式加载图像
image_bgr = cv2.imread("images/plane.jpg", cv2.IMREAD_COLOR)

# 显示像素值
image_bgr[0,0]

array([195, 144, 111], dtype=uint8)
```

一个小提示：OpenCV 默认使用 BGR 格式，但许多图像应用程序（包括 Matplotlib）使用 RGB 格式（红绿蓝），也就是说，红色和蓝色分量的值对调了位置。要在 Matplotlib 中正确显示 OpenCV 彩色图像，需要先将颜色转换为 RGB 格式（抱歉，在黑白印刷的纸质图书中看不出图像有这样的差别）：

```
# 转换为 RGB 格式
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# 显示图像
plt.imshow(image_rgb), plt.axis("off")
plt.show()
```

延伸阅读

- 《RGB 和 BGR 的区别》（*http://bit.ly/2Fws76E*）
- RGB 色彩模型（*http://bit.ly/2FxZjKZ*）

8.2 保存图像

问题描述

为了对图像进行预处理，需要先保存图像。

解决方案

使用 OpenCV 的 `imwrite`：

```
# 加载库
```

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 以灰度图的格式导入图像
image = cv2.imread("images/plane.jpg", cv2.IMREAD_GRAYSCALE)

# 保存图像
cv2.imwrite("images/plane_new.jpg", image)

True
```

讨论

OpenCV 的 `imwrite` 将图像保存到指定的文件路径。图像的格式由文件的扩展名（.jpg，.png 等）决定。需要注意的一点是，`imwrite` 将直接覆盖现有文件，而不会输出一条错误消息或请求你确认。

8.3 调整图像大小

问题描述

为了对图像做进一步的预处理，需要调整图像大小（图像的像素）。

解决方案

使用 `resize` 方法改变图像大小：

```
# 加载图像
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 以灰度图格式导入图像
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# 将图片尺寸调整为 50x50 像素
image_50x50 = cv2.resize(image, (50, 50))

# 查看图像
plt.imshow(image_50x50, cmap="gray"), plt.axis("off")
plt.show()
```

讨论

调整图像大小是图像预处理中常见的任务，原因有两点。首先，原始图像的形状和大小各异，但是被用作特征的图像必须有相同的大小（像素数）。然而图像大小（像素数）的标准化也会带来一些信息上的损失。图像是包含信息的矩阵，当减小图像的像素时，矩阵的尺寸也会缩小，而其中包含的信息也随之减少。其次，机器学习算法可能需要成百上千张图像，甚至更多。如果这些图像非常大，就会占用大量内存，通过调整图像的大小（像素数）可以大大减少内存使用量。在机器学习中常见的图像规格有：32×32、64×64、96×96 和 256×256。

8.4 裁剪图像

问题描述

通过裁剪图像边缘来更改其尺寸。

解决方案

图像被编码为二维 NumPy 数组，因此通过对数组切片可以方便地裁剪图像：

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

```
# 以灰度图格式加载图像
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# 选择所有的行和前 128 列
image_cropped = image[:,:128]

# 显示图像
plt.imshow(image_cropped, cmap="gray"), plt.axis("off")
plt.show()
```

讨论

由于 OpenCV 将图像表示为矩阵，所以选择想要保留的行和列，就可以轻松裁剪图像。如果要保留每张图像的某个部分，这种裁剪方法会特别有用。例如，如果图像来自某个固定的安保摄像头，我们就可以裁剪所有图像，只保留感兴趣的区域。

延伸阅读

- 《NumPy Arrays 切片》（*http://bit.ly/2FrVNBV*）

8.5 平滑处理图像

问题描述

要对图像进行平滑处理。

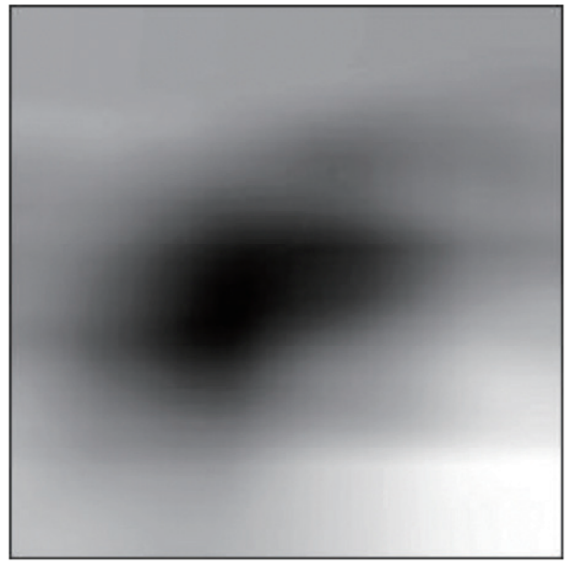
解决方案

平滑处理图像就是将每个像素的值变换为其相邻像素的平均值。相邻像素和所执行的操作在数学上被表示为一个核（如果你不知道核是什么，不用担心）。这个核的大小决定了平滑的程度，核越大，产生的图像就越平滑。这里用一个 5×5 的核对每个像素周围的值取平均值，以此来平滑处理图像：

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 以灰度图格式加载图像
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# 平滑处理图像
image_blurry = cv2.blur(image, (5,5))

# 显示图像
plt.imshow(image_blurry, cmap="gray"), plt.axis("off")
plt.show()
```

为了突出展示核尺寸的影响，下面用 100×100 的核进行相同的平滑操作：

```
# 平滑处理图像
image_very_blurry = cv2.blur(image, (100,100))

# 显示图像
plt.imshow(image_very_blurry, cmap="gray"), plt.xticks([]), plt.yticks([])
plt.show()
```



讨论

在图像处理中，核被广泛应用于从图像锐化到边缘检测的所有领域，本章将反复讨论核的处理。我们使用的平滑核如下所示：

```
# 创建核
kernel = np.ones((5,5)) / 25.0

# 显示核
kernel
```

```
array([[ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04]])
```

核中的中心元素是要处理的像素，而其余元素是该像素的相邻像素。由于所有元素具有相同的值（被归一化为 1），因此每个元素对要处理的像素点有相同的权重。可以使用

filter2D 在图像上手动应用核，以产生与上文类似的平滑效果：

```
# 应用核
image_kernel = cv2.filter2D(image, -1, kernel)

# 显示图像
plt.imshow(image_kernel, cmap="gray"), plt.xticks([]), plt.yticks([])
plt.show()
```

延伸阅读

- 《使用可视化方法解释图像核》（*http://setosa.io/ev/image-kernels/*）
- 《常用的图像核》（*http://bit.ly/2FxZCFD*）

8.6 图像锐化

问题描述

对图像进行锐化处理。

解决方案

创建一个能突出显示目标像素的核，然后使用 filter2D 将其应用于图像：

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 以灰度图格式加载图像
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# 创建核
kernel = np.array([[0, -1, 0],
                   [-1, 5,-1],
                   [0, -1, 0]])

# 锐化图像
image_sharp = cv2.filter2D(image, -1, kernel)

# 显示图像
plt.imshow(image_sharp, cmap="gray"), plt.axis("off")
plt.show()
```

讨论

锐化的原理与平滑相似，在平滑处理中，我们使用核来平均处理相邻像素的值；而在锐化时，我们使用的是能突出显示像素自身的核。锐化可以使图像的边缘更加突出。

8.7 提升对比度

问题描述

增强图像中像素间的对比度。

解决方案

直方图均衡是一种图像处理方法，它可以使图像中的物体和形状更加突出。对于灰度图像，可以直接在图像上应用 OpenCV 的 `equalizeHist`：

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 加载图像
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# 增强图像
image_enhanced = cv2.equalizeHist(image)

# 显示图像
plt.imshow(image_enhanced, cmap="gray"), plt.axis("off")
plt.show()
```

但是如果对彩色图像进行增强操作，首先需要将其转换为 YUV 格式。其中 Y 表示亮度，U 和 V 表示颜色。转换后，可以将 `equalizeHist` 方法应用于此图像，然后将其转换回 BGR 或 RGB 格式：

```
# 加载图像
image_bgr = cv2.imread("images/plane.jpg")

# 转换成 YUV 格式
image_yuv = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2YUV)

# 对图像应用直方图均衡
image_yuv[:, :, 0] = cv2.equalizeHist(image_yuv[:, :, 0])

# 转换成 RGB 格式
image_rgb = cv2.cvtColor(image_yuv, cv2.COLOR_YUV2RGB)

# 显示图像
plt.imshow(image_rgb), plt.axis("off")
plt.show()
```

讨论

关于直方图均衡原理的详细解释超出了本书的范围，简单来说，就是它会转换图像，使像素强度的分布范围更广。

虽然经过直方图均衡后得到的图像往往看起来不“真实”，但是要记住，图像只是底层

数据的可视化表示。如果直方图均衡能够使我们感兴趣的对象与其他对象或背景区分得更明显（并非总是如此），那么它对我们的图像预处理流水线来说就是一个有价值的补充。

8.8 颜色分离

问题描述

将图像中的某种颜色分离出来。

解决方案

定义一个颜色区间，然后对图像应用一个掩模（mask，在图像处理中也称蒙版）。

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 加载图像
image_bgr = cv2.imread('images/plane_256x256.jpg')

# 将 BGR 格式转换成 HSV 格式
image_hsv = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2HSV)

# 定义 HSV 格式中蓝色分量的区间
lower_blue = np.array([50,100,50])
upper_blue = np.array([130,255,255])

# 创建掩模
mask = cv2.inRange(image_hsv, lower_blue, upper_blue)

# 应用掩模
image_bgr_masked = cv2.bitwise_and(image_bgr, image_bgr, mask=mask)

# 从 BGR 格式转换成 RGB 格式
image_rgb = cv2.cvtColor(image_bgr_masked, cv2.COLOR_BGR2RGB)

# 显示图像
plt.imshow(image_rgb), plt.axis("off")
plt.show()
```

讨论

在 OpenCV 中，分离颜色很简单。首先，将图像转换为 HSV 格式（H、S、V 分别代表色调、饱和度、亮度）。其次，定义要分离的一系列值，这部分工作可能是最困难和最耗时的。然后，为图像创建一个掩模（只保留掩模的白色区域）：

```
# 显示图像
plt.imshow(mask, cmap='gray'), plt.axis("off")
plt.show()
```

最后，使用 bitwise_and 方法将掩模应用于图像，并将图像转换为所需的输出格式。

8.9 图像二值化

问题描述

将一张图像转换成仅包含黑色和白色的简化版本。

解决方案

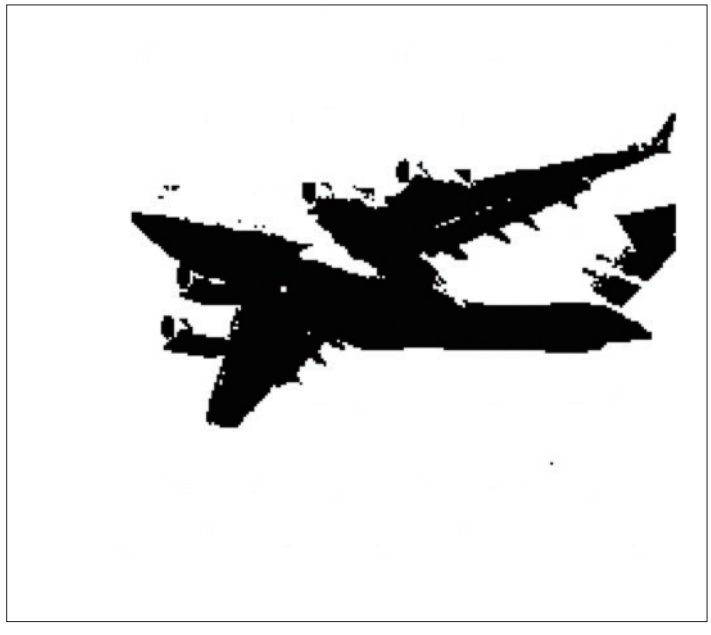
图像二值化是指对图像进行*阈值处理*（thresholding），即将强度大于某个阈值的像素设置为白色并将小于该值的像素设置为黑色的过程。还有一种更先进的技术，叫作*自适应阈值处理*（adaptive thresholding），在这种方法中，一个像素的阈值是由其相邻像素的强度决定的。当图像中不同区域的光照条件有差异时，用这种方法处理会很有帮助。

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 以灰度图像格式加载图像
image_grey = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# 应用自适应阈值处理
max_output_value = 255
neighborhood_size = 99
subtract_from_mean = 10
image_binarized = cv2.adaptiveThreshold(image_grey,
                                        max_output_value,
                                        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                        cv2.THRESH_BINARY,
                                        neighborhood_size,
                                        subtract_from_mean)

# 显示图像
plt.imshow(image_binarized, cmap="gray"), plt.axis("off")
plt.show()
```

讨论

上面调用的 adaptiveThreshold 方法有 4 个重要的参数。参数 max_output_value 用于确定输出像素的最大强度。参数 cv2.ADAPTIVE_THRESH_GAUSSIAN_C 将像素的阈值设置为相邻像素强度的加权和，其权重由高斯窗口确定。我们可以将该参数设置为 cv2.ADAPTIVE_THRESH_MEAN_C，使用相邻像素的平均值作为阈值：

```
# 使用 cv2.ADAPTIVE_THRESH_MEAN_C
image_mean_threshold = cv2.adaptiveThreshold(image_grey,
                                             max_output_value,
                                             cv2.ADAPTIVE_THRESH_MEAN_C,
                                             cv2.THRESH_BINARY,
                                             neighborhood_size,
                                             subtract_from_mean)
# 显示图像
plt.imshow(image_mean_threshold, cmap="gray"), plt.axis("off")
plt.show()
```



最后两个参数分别是块大小（用于确定像素阈值的邻域大小）和一个用来手动调整阈值的常数（从计算出的阈值中减去该常数）。

对图像进行阈值处理的一个主要优点是可以对图像*去噪*——只保留最重要的元素。例如，我们通常对印有文本的照片进行阈值处理，以提取照片上的字母。

8.10 移除背景

问题描述

提取图像中的前景图像。

解决方案

在所需的前景图像周围画一个矩形，然后运行 GrabCut 算法：

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

```
# 加载图像，并将其转换为 RGB 格式
image_bgr = cv2.imread('images/plane_256x256.jpg')
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# 矩形的值：左上角的 x 坐标，左上角的 y 坐标，宽、高
rectangle = (0, 56, 256, 150)

# 创建初始掩模
mask = np.zeros(image_rgb.shape[:2], np.uint8)

# 创建 grabCut 函数需要的临时数组
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)

# 执行 grabCut 函数
cv2.grabCut(image_rgb, # 图像
            mask, # 掩模
            rectangle, # 矩形
            bgdModel, # 背景的临时数组
            fgdModel, # 前景的临时数组
            5, # 迭代次数
            cv2.GC_INIT_WITH_RECT) # 使用定义的矩形初始化

# 创建一个掩模，将确定或很可能是背景的部分设置为 0，其余部分设置为 1
mask_2 = np.where((mask==2) | (mask==0), 0, 1).astype('uint8')

# 将图像与掩模相乘以除去背景
image_rgb_nobg = image_rgb * mask_2[:, :, np.newaxis]

# 显示图像
plt.imshow(image_rgb_nobg), plt.axis("off")
plt.show()
```

讨论

虽然 GrabCut 做得非常好，但是可以看到图像中仍然残留部分背景区域。我们可以手动将这些区域标记为背景，但在现实中往往有数千张图像要处理，逐一手动修改是不可行的。所以，可以考虑接受“留有部分背景噪声”的图像。

在上述解决方案中，首先在包含前景的区域画一个矩形。GrabCut 认为这个矩形之外的一切都是背景，并使用这些信息找出矩形内可能的背景。算法最后会生成一个掩模，它可以标出确定为背景的区域、可能为背景的区域和前景区域：

```
# 显示掩模
plt.imshow(mask, cmap='gray'), plt.axis("off")
plt.show()
```

图中的黑色区域就是我们所画的矩形之外的区域，它被认为是背景。灰色区域是GrabCut 认为的可能为背景的区域，而白色区域则可能是前景图像。

然后，用这个掩模生成一个新的掩模，新掩模会将黑色和灰色区域合并：

```
# 显示掩模
plt.imshow(mask_2, cmap='gray'), plt.axis("off")
plt.show()
```

将第二个掩模应用到图像上，就得到了最终的前景图像。

8.11 边缘检测

问题描述

找出一张图像的边缘。

解决方案

使用 Canny 边缘检测器等边缘检测技术：

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 以灰度图格式加载图像
image_gray = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# 计算像素强度的中位数
median_intensity = np.median(image_gray)

# 设置阈值
lower_threshold = int(max(0, (1.0 - 0.33) * median_intensity))
upper_threshold = int(min(255, (1.0 + 0.33) * median_intensity))

# 应用 Canny 边缘检测器
image_canny = cv2.Canny(image_gray, lower_threshold, upper_threshold)

# 显示图像
plt.imshow(image_canny, cmap="gray"), plt.axis("off")
plt.show()
```

讨论

边缘检测是计算机视觉领域的一个热门话题。边缘很重要，因为它们包含了大量信息。例如，在这张图像中，天空中的一个区域与另一个区域看起来没什么差别，不太可能包含独特或有趣的信息。然而天空与飞机相交的区域却包含大量信息（例如,物体的形状）。边缘检测可以帮助我们去除低信息含量的区域，并将信息含量最高的区域从原始图像中分离出来。

目前有许多边缘检测技术（Sobel 滤波器、Laplacian 边缘检测器等）。上述解决方案使用的是常用的 Canny 边缘检测器。本书不介绍 Canny 边缘检测器的原理，这里仅提醒一点：Canny 边缘检测器需要两个参数来表示梯度的低阈值和高阈值。处于低阈值和高阈值之间的潜在边缘像素被认为是弱边缘像素，而高于阈值的则被认为是强边缘像素。OpenCV 的 `Canny` 方法需要用低阈值和高阈值作为参数。在解决方案中，将这两个阈值分别设置为低于和高于像素强度中位数一个标准偏差的值。如果在对整个图像集运行 Canny 之前，先用一小部分图像试错，手动调整这对阈值，那么通常会得到更好的处理结果。

补充阅读

- Canny 边缘检测器（*http://bit.ly/2FzDXNt*）
- 《自动获取 Canny 边缘检测的阈值》（*http://bit.ly/2nmQERq*）

8.12 角点检测

问题描述

要检测图像角点的位置。

解决方案

使用 OpenCV 实现的 Harris 角点检测器（下文简称 Harris 检测器）`cornerHarris`：

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 以灰度图模式加载图像
image_bgr = cv2.imread("images/plane_256x256.jpg")
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)
image_gray = np.float32(image_gray)

# 设置角点检测器的参数
block_size = 2
aperture = 29
free_parameter = 0.04

# 检测角点
detector_responses = cv2.cornerHarris(image_gray,
                                      block_size,
                                      aperture,
                                      free_parameter)

# 放大角点标志
detector_responses = cv2.dilate(detector_responses, None)

# 只保留大于阈值的检测结果，并把它们标记成白色
threshold = 0.02
image_bgr[detector_responses >
          threshold *
          detector_responses.max()] = [255,255,255]

# 转换成灰度图
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)
```

```
# 显示图像
plt.imshow(image_gray, cmap="gray"), plt.axis("off")
plt.show()
```

讨论

Harris 检测器是检测两条边缘线条的交叉点的常用方法。我们对角点感兴趣的原因与对边缘感兴趣的原因相同：角点含有大量信息。关于 Harris 检测器原理的完整解释可以在本节最后的“延伸阅读”中找到。简单来说，Harris 检测器会寻找一个窗口（也可以叫作邻域），这个窗口的微小移动（可以想象为晃动窗口）会引发窗口内像素值的大幅变化。`cornerHarris` 包含 3 个重要的参数，可以用于控制检测到的边缘。参数 `block_size` 代表角点检测中窗口的尺寸，参数 `aperture` 代表 Sobel 算子的尺寸（如果你不知道什么是 Sobel 算子也没关系），参数 `free_parameter` 用于控制对角点检测的严格程度，这个值越大，可以识别的角点越平滑。

以下程序的输出是一张描绘了可能角点的灰度图像。

```
# 显示可能的角点
plt.imshow(detector_responses, cmap='gray'), plt.axis("off")
plt.show()
```

然后，使用阈值筛选出最可能的角点，或者使用Shi-Tomasi角点检测器（goodFeaturesToTrack）确定一组固定数量的明显的角点，其工作方式与Harris检测器类似。goodFeaturesToTrack有3个主要参数：待检测角点的数量、角点的最差质量（从0到1）以及角点间的最短欧氏距离。

```
# 加载图像
image_bgr = cv2.imread('images/plane_256x256.jpg')
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)

# 待检测角点的数量
corners_to_detect = 10
minimum_quality_score = 0.05
minimum_distance = 25

# 检测角点
corners = cv2.goodFeaturesToTrack(image_gray,
                                  corners_to_detect,
                                  minimum_quality_score,
                                  minimum_distance)
corners = np.float32(corners)

# 在每个角点上画白圈
for corner in corners:
    x, y = corner[0]
    cv2.circle(image_bgr, (x,y), 10, (255,255,255), -1)

# 转换成灰度图
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)
```

```
# 显示图像
plt.imshow(image_rgb, cmap='gray'), plt.axis("off")
plt.show()
```

延伸阅读

- OpenCV 的 `cornerHarris` 方法（*http://bit.ly/2HQXwz6*）
- OpenCV 的 `goodFeaturesToTrack` 方法（*http://bit.ly/2HRSVwF*）

8.13 为机器学习创建特征

问题描述

将图像转换为机器学习算法可用的样本数据。

解决方案

使用 NumPy 的 `flatten` 方法将包含图像信息的多维数组转换成包含样本值的特征向量：

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

```
# 以灰度图格式加载图像
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# 将图像尺寸转换成 10×10
image_10x10 = cv2.resize(image, (10, 10))

# 将图像数据转换成一维向量
image_10x10.flatten()
array([133, 130, 130, 129, 130, 129, 129, 128, 128, 127, 135, 131, 131,
       131, 130, 130, 129, 128, 128, 128, 134, 132, 131, 131, 130, 129,
       129, 128, 130, 133, 132, 158, 130, 133, 130,  46,  97,  26, 132,
       143, 141,  36,  54,  91,   9,   9,  49, 144, 179,  41, 142,  95,
        32,  36,  29,  43, 113, 141, 179, 187, 141, 124,  26,  25, 132,
       135, 151, 175, 174, 184, 143, 151,  38, 133, 134, 139, 174, 177,
       169, 174, 155, 141, 135, 137, 137, 152, 169, 168, 168, 179, 152,
       139, 136, 135, 137, 143, 159, 166, 171, 175], dtype=uint8)
```

讨论

图像是用像素点的网格来表示的。如果是灰度图，则每个像素用一个值表示(即像素强度，白色为 255，黑色为 0)。假设我们有一张 10×10 像素的图像：

```
plt.imshow(image_10x10, cmap="gray"), plt.axis("off")
plt.show()
```

在这种情况下，图像数据的维度是 10×10 ：

```
image_10x10.shape
```

```
(10, 10)
```

如果使用 flatten 方法将数组展开，会得到一个长度为 100（10×10）的向量 ：

```
image_10x10.flatten().shape
```

```
(100,)
```

这就是图像的特征向量，它可以与其他图像的特征向量结合，生成可供机器学习算法使用的数据。

如果是彩色图像，每个像素就不是用一个值表示的了，而是用多个值（最常见的是 3 个），分别表示每个通道（红、绿、蓝等）的强度，这些色彩分量混合后可以表示对应像素点的颜色。因此，如果我们的 10×10 像素的图像是彩色的，每个样本将有 300 个特征值 ：

```
# 以彩色模式加载图像
image_color = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# 将图像尺寸变换为 10×10
image_color_10x10 = cv2.resize(image_color, (10, 10))

 # 将图像数据转换成一维数组并显示数组维度
image_color_10x10.flatten().shape
(300,)
```

由于图像中的每个像素点都是一个特征，所以随着图像尺寸变大，特征的数量也快速增多，这是图像处理和计算机视觉中的一个主要挑战 ：

```
# 以灰度图格式加载图像
image_256x256_gray = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_
GRAYSCALE)

# 将图像转换成一维数组并显示数组维度
image_256x256_gray.flatten().shape
(65536,)
```

如果图像是彩色的，特征的数量还会进一步增加 ：

```
# 以彩色格式加载图像
image_256x256_color = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_
COLOR)
```

```
# 将图像转换成一维数组，并显示数组维度
image_256x256_color.flatten().shape

(196608,)
```

正如输出结果所示，即使是一张小的彩色图像也有近 20 万个特征，在我们训练模型时，这会产生问题，因为特征的数量可能远远超过样本的数量。

这个问题也引发了后面章节对维度策略的讨论：如何在减少特征数量的同时不丢失过多的信息。

8.14 将颜色平均值编码成特征

问题描述

获取一个基于图像颜色的特征。

解决方案

彩色图像中的每个像素都是用多个颜色通道（通常为 3 个：红、绿、蓝）的组合来表示的。计算图像每种颜色通道（红、绿、蓝）的平均值，就可以得到代表该图像颜色均值的三种颜色特征：

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 以 BGR 格式加载图像
image_bgr = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# 计算每个通道的平均值
channels = cv2.mean(image_bgr)

# 交换红色和蓝色通道，将图像从 BGR 格式转换成 RGB 格式
observation = np.array([(channels[2], channels[1], channels[0])])

# 显示每个颜色通道的平均值
observation
array([[  90.53204346,  133.11735535,  169.03074646]])
```

可以直接显示各颜色通道的平均值（在黑白印刷的纸质书上看不到实际的效果）：

```
# 显示图像
plt.imshow(observation), plt.axis("off")
plt.show()
```

讨论

本节代码的输出结果是样本的三个特征值，分别来自图像的各颜色通道。这些特征可以和其他特征一样用于机器学习算法，根据颜色对图像进行分类。

8.15 将色彩直方图编码成特征

问题描述

创建一组特征来表示图像中的颜色。

解决方案

计算每一个颜色通道的直方图：

```
# 加载库
import cv2
import numpy as np
from matplotlib import pyplot as plt

# 加载图像
image_bgr = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# 将图像转换成 RGB 格式
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# 创建一个特征列表
features = []
```

```
# 为每一个颜色通道计算直方图
colors = ("r","g","b")

# 为每一个通道计算直方图并把它加入特征列表中
for i, channel in enumerate(colors):
    histogram = cv2.calcHist([image_rgb], # 图像
                        [i], # 颜色通道的序号
                        None, # 不使用掩模
                        [256], # 直方图尺寸
                        [0,256]) # 范围
    features.extend(histogram)

# 将样本的特征值展开成一维数组
observation = np.array(features).flatten()

# 显示样本的前 5 个特征值
observation[0:5]
array([ 1008.,   217.,   184.,   165.,   116.], dtype=float32)
```

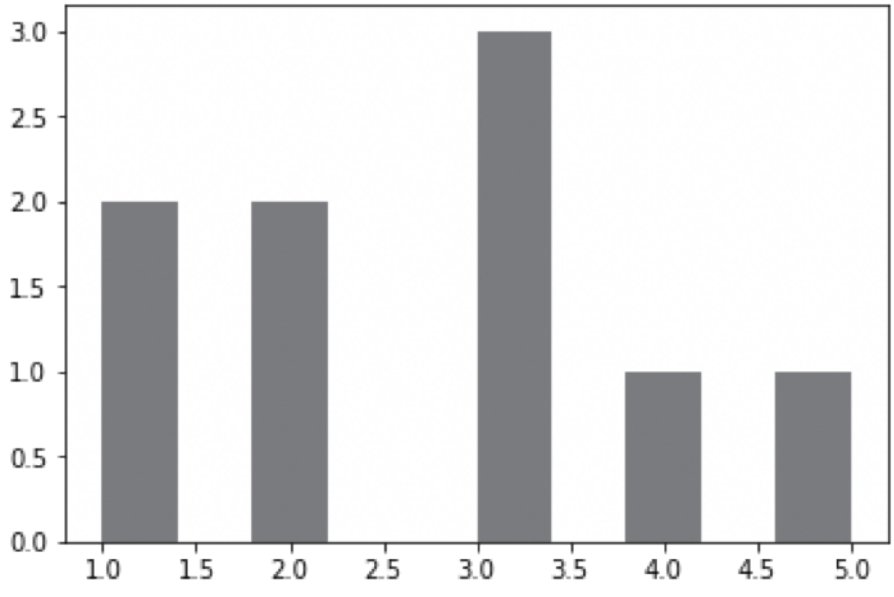
讨论

在 RGB 色彩模型中，每种颜色是由三个通道（即红、绿、蓝）的值组合而成的。每个通道可以取 256 个值中的一个（0 到 255 之间的整数）。例如，图像左上角像素的通道值如下：

```
# 显示 RGB 通道的值
image_rgb[0,0]
array([107, 163, 212], dtype=uint8)
```

直方图是一种表示数据的值分布的方法。下面是一个简单的例子：

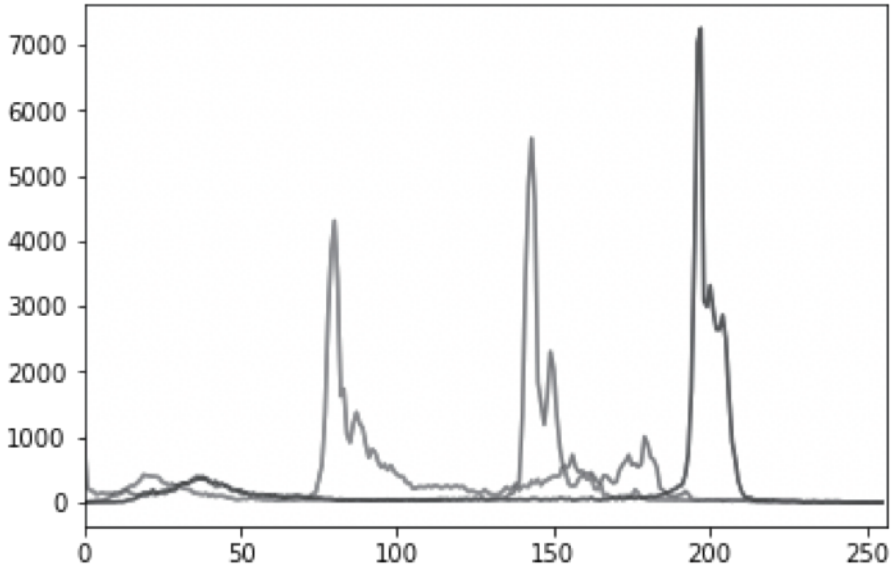
```
# 导入 pandas
import pandas as pd

# 创建一些数据
data = pd.Series([1, 1, 2, 2, 3, 3, 3, 4, 5])

# 显示直方图
data.hist(grid=False)
plt.show()
```



在这个例子中，我们的数据中有两个 1、两个 2、三个 3、一个 4 和一个 5。在直方图中，每个竖条代表一个值（例如 1、2 等）在数据中出现的次数。

可以将这种方法应用于每个颜色通道，不过这里可能的值不是 5 个，而是 256 个（通道值的范围）。*x* 轴表示 256 个可能的通道值，*y* 轴表示某个特定的通道值在图像中出现的次数。

```
# 计算每个颜色通道的直方图
colors = ("r","g","b")

# 对每个通道绘制直方图
for i, channel in enumerate(colors):
    histogram = cv2.calcHist([image_rgb], # 图像
                        [i], # 颜色通道的序号
                        None, # 不使用掩模
                        [256], # 直方图尺寸
                        [0,256]) # 范围
    plt.plot(histogram, color = channel)
    plt.xlim([0,256])

# 显示图
plt.show()
```



正如我们在直方图中看到的，几乎没有像素的蓝色通道值在0到180之间，许多像素的蓝色通道值在190到210之间。三个通道所有值的分布都在图中显示出来了。但是该图不仅仅是对数据的可视化，还显示了每个颜色通道的256个特征值，总共768个特征值，代表了图像中颜色的分布。

延伸阅读

- 直方图（*https://en.wikipedia.org/wiki/Histogram*）
- pandas 关于直方图的文档（*http://bit.ly/2HT4Fz0*）
- OpenCV 的直方图教程（*http://bit.ly/2HSyoYH*）

第 9 章

利用特征提取进行特征降维

9.0 简介

通常情况下，一幅图像拥有成千上万的特征。例如，在第 8 章中曾将一幅 256×256 像素的彩色图像转换为 196,608 个特征。而且，由于每个像素都可以取 256 个可能值中的任一个，所以每个样本（即一幅彩色像素图像）会有 256^{196608} 种不同的像素值组合。由于在实际操作中很难收集到足够多的有效样本，导致机器学习算法没有足够的数据进行训练，这给整个机器学习过程带来很大困难。

幸运的是，并非所有的特征都是同样重要的，通过特征提取（feature extraction）进行降维的目的就是对原特征集 $P_{original}$ 进行变换，变换后得到的新特征集记为 P_{new}，这里 $P_{original}>P_{new}$，但是 P_{new} 保留了原特征集的大部分信息。换句话说，就是通过牺牲一小部分数据信息来减少特征的数量，并且保证还能做出准确的预测。本章将介绍一组做特征提取的方法。

上面所讨论的特征提取技术存在一个缺点，即产生的新特征对人类而言是不可解释的。新特征拥有训练模型的能力，但是在我们人类看来它们却像是一些随机数字的集合。如果你希望能解释模型，可以采用特征选择来进行特征降维。

9.1 使用主成分进行特征降维

问题描述

对于给定的一组特征，在保留信息量的同时减少特征的数量。

解决方案

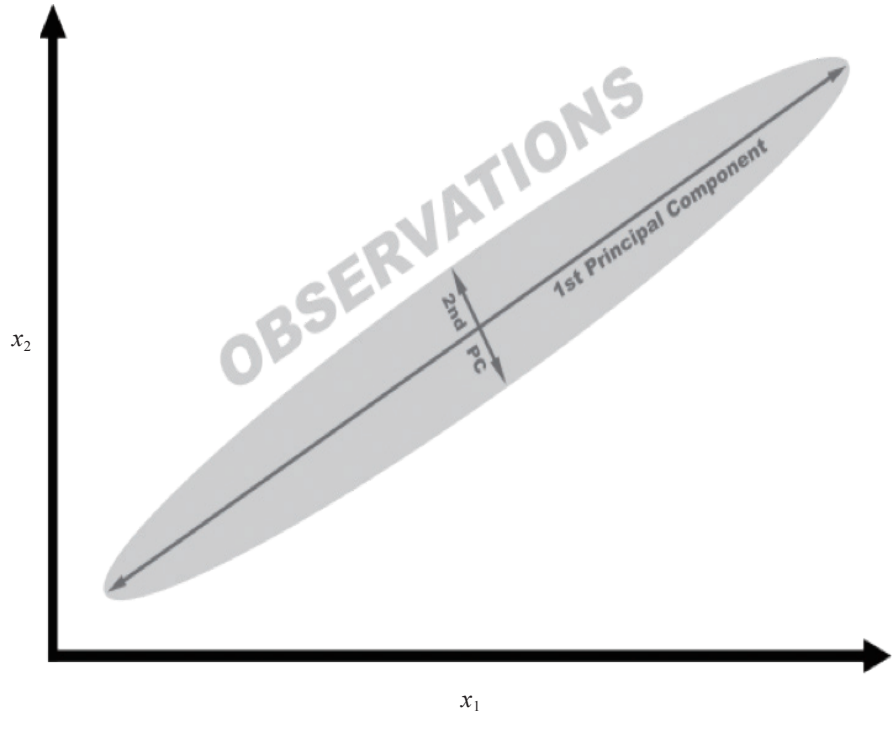
使用 scikit-learn 库中的主成分分析工具（PCA）：

```
# 加载库
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import datasets

# 加载数据
digits = datasets.load_digits()

# 标准化特征矩阵
features = StandardScaler().fit_transform(digits.data)

# 创建可以保留 99% 信息量（用方差表示）的 PCA
pca = PCA(n_components=0.99, whiten=True)

# 执行 PCA
features_pca = pca.fit_transform(features)

# 显示结果
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_pca.shape[1])

Original number of features: 64
Reduced number of features: 54
```

讨论

主成分分析法（Principal Component Analysis，PCA）是一种流行的线性降维方法。PCA 将样本数据映射到特征矩阵的主成分空间（主成分空间保留了大部分的数据差异，一般具有更低的维度）。PCA 是一种无监督学习方法，也就是说它只考虑特征矩阵而不需要目标向量的信息。

在本节最后的“延伸阅读”中列出了关于 PCA 原理的数学描述的文章。这里我们用一个简单的例子来介绍 PCA 的原理。下图的数据包含两个特征 x_1 和 x_2。从图中可以清楚地看到，样本的分布像一支侧放的雪茄烟，其长度远大于高度。在这里，可以用主成分（Principal Component）的方向来代替长度和高度，用特征变化最大的方向作为第一个主成分，变化第二大的方向作为第二个主成分，依此类推。



如果想降低特征的维度，可以将二维空间的样本映射到一维的主成分空间。这样做会损失第二个主成分携带的一些信息，你需要做出权衡，但在某些情况下这是可以接受的。这就是 PCA 的原理。

在 scikit-learn 中，PCA 由方法 `pca` 实现。参数 `n_components` 有两种含义，由具体的参数值决定。如果值大于 1，则 `n_components` 将会返回和这个值相同数量的特征，但这也带来一个问题，即如何选择最合适的特征数量。幸运的是，如果 `n_components` 的值在 0 和 1 之间，`pca` 就会返回维持一定信息量（在算法中，用方差代表信息量）的最少特征数。通常情况下，`n_components` 取值为 0.95 或 0.99，意味着保留 95% 或 99% 的原始特征信息量。参数 `whiten=True`,表示对每一个主成分都进行转换以保证它们的平均值为 0、方差为 1。另一个参数是 `svd_solver="randomized"`，代表使用随机方法找到第一个主成分（这种方法通常速度很快）。

解决方案的输出结果显示，PCA 保留了特征矩阵 99% 的信息，同时将特征的数量减少了 10 个。

延伸阅读

- 关于 PCA 的 scikit-learn 文档（*http://bit.ly/2FrSvyx*）

- 《选择主成分的数量》（*http://bit.ly/2FrSGtH*）
- 《使用线性代数进行主成分分析》（*http://bit.ly/2FuzdIW*）

9.2 对线性不可分数据进行特征降维

问题描述

对线性不可分数据进行特征降维。

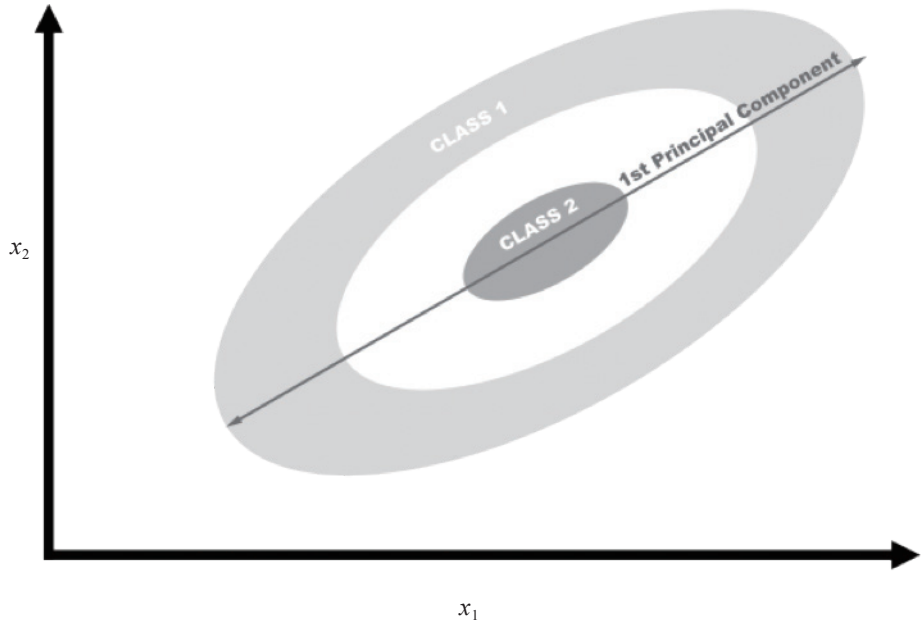
解决方案

使用 Kernel PCA（核主成分分析，PCA 的一种扩展）进行非线性降维：

```
    # 加载库
    from sklearn.decomposition import PCA, KernelPCA
    from sklearn.datasets import make_circles

    # 创建线性不可分数据
    features, _ = make_circles(n_samples=1000, random_state=1, noise=0.1,
factor=0.1)

    # 应用基于径向基函数（Radius Basis Function，RBF）核的 Kernel PCA 方法
    kpca = KernelPCA(kernel="rbf", gamma=15, n_components=1)
    features_kpca = kpca.fit_transform(features)

    print("Original number of features:", features.shape[1])
    print("Reduced number of features:", features_kpca.shape[1])

    Original number of features: 2
    Reduced number of features: 1
```

讨论

PCA 能够降低特征矩阵的维度（例如，减少特征的数量）。标准 PCA 使用线性映射减少特征的数量。如果数据是线性可分的（也就是说，可以用一条直线或者超平面将两类数据分开），那么 PCA 的处理效果会很好。然而，如果数据不是线性可分的（例如，只能用曲线边界分类的数据），那么线性变换的效果不会很好。在上面的解决方案中，我们使用 scikit-learn 库的 `make_circles` 方法生成了一个模拟数据集，有一个拥有两个分类（`class`）和两个特征的目标向量。`make_circles` 生成的就是线性不可分数据，其中一个分类被另一个分类从外部完全包围（即环形数据，参见下图）。



如果使用线性 PCA 对数据进行降维，则两类数据会被线性映射到第一个主成分（Principal Component）上，因而会交织在一起，参见下图。

理想情况下，我们希望维度变换既能降低数据的维度，又可以使数据变得线性可分。Kernel PCA 可以做到这两点，参见下图。

核（kernel，也叫核函数）能够将线性不可分数据映射到更高的维度，数据在这个维度是线性可分的，我们把这种方式叫作核机制（kernel trick）。如果你不懂核机制的细节，不用担心，就把核机制看作映射数据的一种方法吧。scikit-learn 的 `KernelPCA` 中提供了很多可用的核函数，可以使用参数 `kernel` 来指定。常用的核函数是高斯径向基函数（`rbf`），其他的核函数还有多项式核（`poly`）和 sigmoid 核（`sigmoid`）。我们甚至可以指定一个线性（`linear`）映射，利用它可以得到与标准 PCA 相同的结果。

`KernelPCA` 的一个缺点是，需要指定很多参数。例如在 9.1 节中，我们将参数 `n_components` 设置为 0.99，便可以让 `PCA` 选择保留 99% 信息量的特征数，然而在

KernelPCA 中没有这个选项。KernelPCA 必须指定参数的数量(例如,n_components = 1),此外,每个核都有自己的超参数需要设置,例如,径向基函数需要设置伽马值(gamma)。

那么,如何设置这些值呢?可以通过反复试错来确定。也就是说,使用不同的核函数和参数值反复训练机器学习模型,找出产生最优模型的参数组合。我们将在第 12 章深入学习这种方法。

延伸阅读

- scikit-learn 文档:Kernel PCA(*http://bit.ly/2HRkxC3*)
- 核技巧及使用 RBF 核函数的非线性降维(*http://bit.ly/2HReP3f*)

9.3 通过最大化类间可分性进行特征降维

问题描述

对特征进行降维操作,然后将其应用于分类器。

解决方案

使用线性判别分析(Linear Discriminant Analysis,LDA)方法,将特征数据映射到一个可以使类间可分性最大的成分坐标轴上。

```
# 加载库
from sklearn import datasets
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# 加载 Iris flower 数据集
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建并运行 LDA,然后用它对特征做变换
lda = LinearDiscriminantAnalysis(n_components=1)
features_lda = lda.fit(features, target).transform(features)

# 打印特征的数量
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_lda.shape[1])

Original number of features: 4
Reduced number of features: 1
```

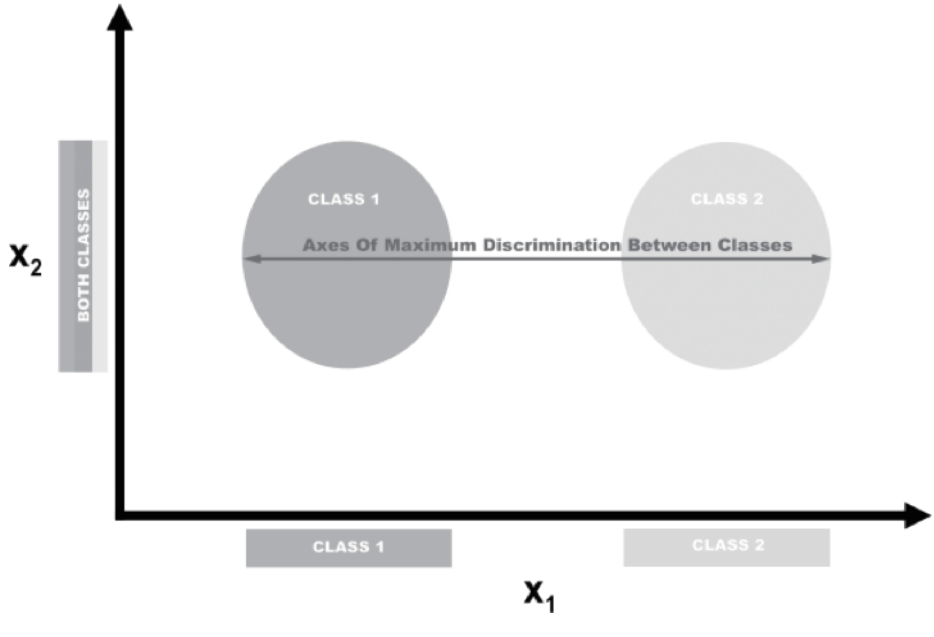
可以使用参数 explained_variance_ratio_ 来查看每个成分保留的信息量（即数据的差异）的情况。在本解决方案中，单个成分保留了 99% 的信息量：

```
lda.explained_variance_ratio_
```

```
array([ 0.99147248])
```

讨论

LDA 是一种分类方法，也是常用的降维方法。LDA 与 PCA 的原理类似，它将特征空间映射到较低维的空间。然而在 PCA 中，只需关注使数据差异最大化的成分轴；在 LDA 中，我们的另一个目标是，找到使得类间差异最大的成分轴。在下图所示的例子中，数据集由具有两个分类、两个特征的数据构成。如果将数据投影到 y 轴上，那么两类数据将不易分离（互相重叠）；而如果将数据投影到 x 轴上，则只剩下一个特征向量（即降低了一个维度），并且仍然能保持类的可分性。当然，在现实世界中，分类之间的关系会更加复杂，维度也会更高，但概念是一样的。



在 scikit-learn 中，LDA 是由 LinearDiscriminantAnalysis 方法实现的，其中包含一个参数 n_components，表示需要返回的特征数量。为了找出 n_components 的最优值（例如，需要保留多少个参数），我们可以参考 explain_variance_ratio_ 的输出，该排序数组表示每个输出的特征所保留的信息量（用方差表示），例如：

```
lda.explained_variance_ratio_
array([ 0.99147248])
```

具体来说，可以运行 LinearDiscriminantAnalysis（将参数 n_components 设置为 None）返回每个成分特征保留的信息量的百分比，然后计算需要多少个成分特征才能保留高于阈值（通常为 0.95 或 0.99）的信息量：

```
# 创建并运行 LDA
lda = LinearDiscriminantAnalysis(n_components=None)
features_lda = lda.fit(features, target)

# 获取方差的百分比数组
lda_var_ratios = lda.explained_variance_ratio_

# 函数定义
def select_n_components(var_ratio, goal_var: float) -> int:
    # 设置总方差的初始值
    total_variance = 0.0

    # 设置特征数量的初始值
    n_components = 0

    # 遍历方差百分比数组的元素
    for explained_variance in var_ratio:

        # 将该百分比加入总方差
        total_variance += explained_variance

        # n_components 的值加 1
        n_components += 1

        # 如果达到目标阈值
        if total_variance >= goal_var:
            # 结束遍历
            break

    # 返回 n_components 的值
    return n_components

# 运行函数
select_n_components(lda_var_ratios, 0.95)

1
```

延伸阅读

- 比较 LDA 和 PCA 在 Iris 数据集上的二维映射（*http://bit.ly/2Fs4cWe*）
- 《线性判别分析》（*http://bit.ly/2FtiKEL*）

9.4 使用矩阵分解法进行特征降维

问题描述

对非负特征矩阵进行降维。

解决方案

使用非负矩阵分解法（Non-Negative Matrix Factorization，NMF）对特征矩阵进行降维。

```
# 加载库
from sklearn.decomposition import NMF
from sklearn import datasets

# 加载数据
digits = datasets.load_digits()

# 加载特征矩阵
features = digits.data

# 创建 NMF，进行变换并应用
nmf = NMF(n_components=10, random_state=1)
features_nmf = nmf.fit_transform(features)

# 显示结果
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_nmf.shape[1])
```

```
Original number of features: 64
Reduced number of features: 10
```

讨论

NMF 是一种无监督的线性降维方法，它可以分解矩阵（将特征矩阵分解为多个矩阵，其乘积近似于原始矩阵），将特征矩阵转换为表示样本与特征之间潜在关系的矩阵。简单地说，NMF 可以减少维数，因为在矩阵乘法中，两个因子（相乘的矩阵）的维数要比得到的乘积矩阵的维数低得多。正式地，给定一个期望的返回特征的数量，NMF 将把特征

矩阵分解为：

$$\boldsymbol{V} \approx \boldsymbol{WH}$$

其中，$\boldsymbol{V}$ 是 $d \times n$ 维特征矩阵（即 d 个特征，n 个样本），$\boldsymbol{W}$ 为 $d \times r$ 维矩阵，$\boldsymbol{H}$ 是 $r \times n$ 维矩阵。通过调整值，可以设定希望减少的维数。

如果要使用 NMA，特征矩阵就不能包含负数值。此外，与 PCA 等前面探讨过的技术不同，NMA 不会告诉我们输出特征中保留了原始数据的信息量。因此，找出参数 `n_components` 的最优值的最佳方法是不断尝试一系列可能的值，直到找出能生成最佳学习模型的值（见第 12 章）。

延伸阅读

- 非负矩阵分解（NMF）（*http://bit.ly/2FvtWRj*）

9.5 对稀疏数据进行特征降维

问题描述

对稀疏特征矩阵进行特征降维操作。

解决方案

使用截断奇异值分解（Truncated Singular Value Decomposition，TSVD）法：

```
# 加载多个库
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import TruncatedSVD
from scipy.sparse import csr_matrix
from sklearn import datasets
import numpy as np

# 加载数据
digits = datasets.load_digits()

# 标准化特征矩阵
features = StandardScaler().fit_transform(digits.data)

# 生成稀疏矩阵
features_sparse = csr_matrix(features)
```

```
# 创建 tsvd
tsvd = TruncatedSVD(n_components=10)

# 在稀疏矩阵上执行 TSVD
features_sparse_tsvd = tsvd.fit(features_sparse).transform(features_sparse)

# 显示结果
print("Original number of features:", features_sparse.shape[1])
print("Reduced number of features:", features_sparse_tsvd.shape[1])

Original number of features: 64
Reduced number of features: 10
```

讨论

TSVD 与 PCA 类似，事实上 PCA 常常在某一个步骤中使用非截断奇异值分解（SVD）法。在常规 SVD 中，对于给定的 d 个特征，SVD 将创建 $d \times d$ 维的因子矩阵，而 TSVD 将返回 $n \times n$ 维的因子矩阵，其中 n 是预先指定的参数。与 PCA 相比，TSVD 的优势在于它适用于稀疏特征矩阵。

但是 TSVD 有一个问题，其输出值的符号会在多次拟合中不断变化（这是由其使用随机数生成器的方式决定的）。一个简单的解决方法是对每个预处理管道只使用一次 fit 方法，然后多次使用 transform 方法。

与 LDA 一样，TSVD 需要通过参数 n_components 指定想要输出的特征（成分）数。那么，输出的特征数是多少时可以保证效果最佳？寻找最佳特征数的一种方法是在模型选择时将 n_components 作为超参数进行优化（即选择产生最佳训练模型的 n_components 值）。由于 TSVD 提供了每个成分保留的原始特征矩阵信息的比例，因而我们也可以按照要保留的信息量（常用的值是 95%或 99%）选择成分。在本解决方案中，前三个输出的成分能够保留大约 30%的原始数据信息：

```
# 对前三个成分的信息量占比求和
tsvd.explained_variance_ratio_[0:3].sum()

0.30039385386597783
```

可以创建一个运行 TSVD 的函数，使这个过程自动化（将参数 n_components 设置为原始特征数量减 1），然后计算能够保留所需信息量的特征数量。

```
# 用比原特征数量小 1 的值作为 n_components 的值，创建并运行 TSVD
tsvd = TruncatedSVD(n_components=features_sparse.shape[1]-1)
features_tsvd = tsvd.fit(features)
```

```
# 获取方差百分比数组
tsvd_var_ratios = tsvd.explained_variance_ratio_

# 函数定义
def select_n_components(var_ratio, goal_var):
    # 设置总方差的初始值
    total_variance = 0.0

    # 设置特征数量的初始值
    n_components = 0

    # 遍历方差百分比数组的元素
    for explained_variance in var_ratio:

        # 将该百分比加入总方差
        total_variance += explained_variance

        # n_components 的值加 1
        n_components += 1

        # 如果达到目标阈值
        if total_variance >= goal_var:
            # 结束遍历
            break

    # 返回 n_components 的值
    return n_components

# 运行函数
select_n_components(tsvd_var_ratios, 0.95)

40
```

延伸阅读

- scikit-learn 文档：TruncatedSVD（*http://bit.ly/2GTfxvh*）

第 10 章

使用特征选择进行降维

10.0 简介

第 9 章讨论了如何在尽可能不丢失特征原始信息的情况下，通过创建新特征来降低特征矩阵的维度，这种方法被称为特征提取（feature extraction）。本章将介绍另一种降维方法：特征选择（feature selection）。特征选择会保留信息量较高的特征而丢弃信息量较低的特征。

特征选择的方法可分为三类：过滤器、包装器和嵌入式方法。过滤器方法根据特征的统计信息来选择最优特征。包装器方法通过不断试错，找出一个可以产生高质量预测值的模型的特征子集。嵌入式方法则将选择最优特征子集作为机器学习算法训练过程的一部分。

按理来说，我们应该在本章介绍这三种方法，但是嵌入式方法与特定的机器学习算法密切相关，在深入研究这些算法之前很难把嵌入式方法解释清楚。因此，本章只讲解过滤器和包装器这两种特征选择方法，而把对嵌入式方法的介绍放到深入讨论机器学习算法的章节。

10.1 数值型特征方差的阈值化

问题描述

从一组数值型特征中移除方差较小（即可能包含的信息较少）的特征。

解决方案

挑出方差大于给定阈值的特征：

```
# 加载库
from sklearn import datasets
from sklearn.feature_selection import VarianceThreshold

# 加载数据
iris = datasets.load_iris()

# 创建 features 和 target
features = iris.data
target = iris.target

# 创建 VarianceThreshold 对象
thresholder = VarianceThreshold(threshold=.5)

# 创建大方差特征矩阵
features_high_variance = thresholder.fit_transform(features)

# 显示大方差特征矩阵
features_high_variance[0:3]

array([[ 5.1,  1.4,  0.2],
       [ 4.9,  1.4,  0.2],
       [ 4.7,  1.3,  0.2]])
```

讨论

方差阈值化（Variance Thresholding，VT）是最基本的特征选择方法之一。这种方法的依据是小方差的特征可能比大方差的特征的重要性低一些。VT 方法的第一步是计算每个特征的方差：

$$\text{operatornameVar}(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

其中 x 是特征向量，x_i 代表一个单独的特征值，μ 是特征的均值。算出方差后，方差低于阈值的特征会被丢弃。

采用 VT 方法时需要注意两点。第一，方差不是中心化的（它的单位是特征单位的平方）。因此，如果特征数据集中特征的单位不同（例如，一个特征以年为单位，而另一个特征以美元为单位），那么 VT 法就无法起作用。第二，方差的阈值是手动选择的，所以必须依靠人工来选择一个合适的阈值（或者使用第 12 章中的模型选择方法）。可以通过参数 `variances_` 来查看每个特征的方差：

```
# 显示方差
thresholder.fit(features).variances_
array([ 0.68112222,  0.18675067,  3.09242489,  0.57853156])
```

最后，如果特征已经标准化（即平均值为 0，方差为 1），方差阈值将起不到筛选的作用（因为这时特征的方差全部为 1）：

```
# 加载库
from sklearn.preprocessing import StandardScaler

# 标准化特征矩阵
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# 计算每个特征的方差
selector = VarianceThreshold()
selector.fit(features_std).variances_
array([ 1.,  1.,  1.,  1.])
```

10.2 二值特征的方差阈值化

问题描述

有一组二值特征数据（即只有两个分类），现在要移除其中方差较小的特征。

解决方案

挑出方差大于给定阈值的二值特征：

```
# 加载库
from sklearn.feature_selection import VarianceThreshold

# 用如下信息创建特征矩阵：
# 特征 0: 80% 为分类 0
# 特征 1: 80% 为分类 1
# 特征 2: 60% 为分类 0, 40% 为分类 1
features = [[0, 1, 0],
            [0, 1, 1],
            [0, 1, 0],
            [0, 1, 1],
            [1, 0, 0]]

# 创建 VarianceThreshold 对象并运行
thresholder = VarianceThreshold(threshold=(.75 * (1 - .75)))
thresholder.fit_transform(features)

array([[0],
       [1],
       [0],
       [1],
       [0]])
```

讨论

和数值型特征一样，挑选高信息量的分类特征的方法之一就是查看它们的方差。在二值特征（即伯努利随机变量）中，方差的计算公式如下：

$$\text{Var}(x)=p(1-p)$$

其中，p 是观察值属于第 1 个分类的概率。通过设置 p 的值，我们可以删除大部分观察值都属于同一个类别的特征。

10.3 处理高度相关性的特征

问题描述

特征矩阵中的某些特征具有较高的相关性。

解决方案

使用相关矩阵检查是否存在较高相关性的特征，如果存在，则删除其中的一个：

```
# 加载库
import pandas as pd
import numpy as np

# 创建一个特征矩阵，其中包含两个高度相关的特征
features = np.array([[1, 1, 1],
                     [2, 2, 0],
                     [3, 3, 1],
                     [4, 4, 0],
                     [5, 5, 1],
                     [6, 6, 0],
                     [7, 7, 1],
                     [8, 7, 0],
                     [9, 7, 1]])

# 将特征矩阵转换成 DataFrame
dataframe = pd.DataFrame(features)

# 创建相关矩阵
corr_matrix = dataframe.corr().abs()

# 选择相关矩阵的上三角阵
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape),
```

```
                                k=1).astype(np.bool))

# 找到相关性大于 0.95 的特征列的索引
to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]

# 删除特征
dataframe.drop(dataframe.columns[to_drop], axis=1).head(3)
```

	0	2
0	1	1
1	2	0
2	3	1

讨论

在机器学习中经常会遇到特征高度相关的问题。如果两个特征高度相关，那么它们所包含的信息就非常相似，因此这两个特征就存在冗余。解决这个问题的方法很简单：从特征集中删除一个与其他特征高度相关的特征即可。

在上述解决方案中，首先创建一个所有特征的相关矩阵：

```
# 相关矩阵
dataframe.corr()
```

	0	1	2
0	1.000000	0.976103	0.000000
1	0.976103	1.000000	-0.034503
2	0.000000	-0.034503	1.000000

然后，从相关矩阵的上三角阵中找出相关性高的特征对：

```
# 相关矩阵的上三角阵
upper
```

	0	1	2
0	NaN	0.976103	0.000000
1	NaN	NaN	0.034503
2	NaN	NaN	NaN

最后，从每一对高度相关的特征中删除一个。

10.4 删除与分类任务不相关的特征

问题描述

根据分类的目标向量，删除信息量较低的特征。

解决方案

对于分类型特征，计算每个特征和目标向量的卡方统计量（χ^2，chi-squared）：

```
# 加载库
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2, f_classif

# 加载数据
iris = load_iris()
features = iris.data
target = iris.target

# 将分类数据转换成整数型数据
features = features.astype(int)

# 选择卡方统计量最大的两个特征
chi2_selector = SelectKBest(chi2, k=2)
features_kbest = chi2_selector.fit_transform(features, target)

# 显示结果
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])
```

```
Original number of features: 4
Reduced number of features: 2
```

对于数值型特征数据，则计算每个特征与目标向量之间的方差分析 F 值（ANOVA F-value）：

```
# 选择 F 值最大的两个特征
fvalue_selector = SelectKBest(f_classif, k=2)
features_kbest = fvalue_selector.fit_transform(features, target)

# 显示结果
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])
```

```
Original number of features: 4
Reduced number of features: 2
```

除了选择固定数量的特征，还可以通过 `SelectPercentile` 方法来选择前 n% 的特征：

```
# 加载库
from sklearn.feature_selection import SelectPercentile

# 选择 F 值位于前 75% 的特征
fvalue_selector = SelectPercentile(f_classif, percentile=75)
features_kbest = fvalue_selector.fit_transform(features, target)

# 显示结果
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])

Original number of features: 4
Reduced number of features: 3
```

讨论

卡方统计可以检查两个分类向量的相互独立性。也就是说，卡方统计量代表了观察到的样本数量和期望的样本数量（假设特征与目标向量无关）之间的差异。

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

其中，O_i 是第 i 个分类的样本数量；E_i 是假设特征 i 和目标向量无关时，我们期望的第 i 个分类的样本数量。

卡方统计的结果是一个数字，它代表观察值和期望值（假设特征与目标向量无关时的值）之间的差异。通过计算特征和目标向量的卡方统计量，我们可以得到对两者之间独立性的度量值。如果两者相互独立，就代表特征与目标向量无关，特征中不包含分类所需要的信息。相反，如果特征和目标向量高度相关，则代表特征中包含训练分类模型所需的很多信息。

为了使用卡方统计进行特征选择，首先需要计算每个特征和目标向量之间的卡方统计量，然后选择卡方统计量最大的特征。在 scikit-learn 中，可以使用 `SelectKBest` 方法来选择最好的特征。其中的参数 `k` 决定了要保留的特征数量。

需要注意的是，卡方统计量只能在两个分类数据之间进行计算，所以使用卡方统计进行特征选择时，目标向量和特征都必须是分类数据。对于数值型特征，可以将其转换为分类特征后再使用卡方统计。最后，使用卡方统计方法时，所有值都必须是非负的。

除此之外，对于数值型特征还可以使用 `f_classif` 来计算每个特征和目标向量间的

ANOVA F 值。在根据目标向量对数值型特征分类时，该值可以用来判断每个分类的特征均值之间的差异有多大。例如，如果有一个二元目标向量（性别）和一个数值型特征（考试分数），那么 ANOVA F 值可以用来判断男性的平均得分是否与女性的相同。如果相同，那么考试分数并不能帮助我们预测性别，因此这个特征与目标向量是无关的。

10.5 递归式特征消除

问题描述

自动选择需要保留的最优特征。

解决方案

使用 scikit-learn 的 RFECV 类通过交叉验证（Crossing Validation，CV）进行递归式特征消除（Recursive Feature Elimination，REF）。该方法会重复训练模型，每一次训练移除一个特征，直到模型性能（例如精度）变差。剩余的特征就是最优特征：

```
# 加载库
import warnings
from sklearn.datasets import make_regression
from sklearn.feature_selection import RFECV
from sklearn import datasets, linear_model

# 忽略一些烦人但无害的警告信息
warnings.filterwarnings(action="ignore", module="scipy",
                        message="^internal gelsd")

# 生成特征矩阵、目标向量以及系数
features, target = make_regression(n_samples = 10000,
                                   n_features = 100,
                                   n_informative = 2,
                                   random_state = 1)
# 创建线性回归对象
ols = linear_model.LinearRegression()

# 递归消除特征
rfecv = RFECV(estimator=ols, step=1, scoring="neg_mean_squared_error")
rfecv.fit(features, target)
rfecv.transform(features)
```

```
array([[ 0.00850799,  0.7031277 , -1.2416911 , -0.25651883, -0.10738769],
       [-1.07500204,  2.56148527,  0.5540926 , -0.72602474, -0.91773159],
       [ 1.37940721, -1.77039484, -0.59609275,  0.51485979, -1.17442094],
       ...,
       [-0.80331656, -1.60648007,  0.37195763,  0.78006511, -0.20756972],
       [ 0.39508844, -1.34564911, -0.9639982 ,  1.7983361 , -0.61308782],
       [-0.55383035,  0.82880112,  0.24597833, -1.71411248,  0.3816852 ]])
```

执行完 RFE 后，就可以得到应该保留的特征数量：

```
# 最优特征的数量
rfecv.n_features_

5
```

也可以查看哪些特征应该被保留：

```
# 哪些特征是最优特征
rfecv.support_

array([False, False, False, False, False,  True, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False,  True, False, False, False,  True, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False,  True, False, False,
       False, False, False, False, False,  True, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False], dtype=bool)
```

甚至可以查看特征的排名：

```
# 将特征从最好（1）到最差排序
rfecv.ranking_

array([11, 92, 96, 87, 46,  1, 48, 23, 16,  2, 66, 83, 33, 27, 70, 75, 29,
       84, 54, 88, 37, 42, 85, 62, 74, 50, 80, 10, 38, 59, 79, 57, 44,  8,
       82, 45, 89, 69, 94,  1, 35, 47, 39,  1, 34, 72, 19,  4, 17, 91, 90,
       24, 32, 13, 49, 26, 12, 71, 68, 40,  1, 43, 63, 28, 73, 58, 21, 67,
        1, 95, 77, 93, 22, 52, 30, 60, 81, 14, 86, 18, 15, 41,  7, 53, 65,
       51, 64,  6,  9, 20,  5, 55, 56, 25, 36, 61, 78, 31,  3, 76])
```

讨论

这可能是到目前为止本书所讲的最先进的解决方案，涉及一些我们尚未详细讨论的内容。不过它的思路很简单，可以直接在这里解释。 RFE 背后的思想是重复训练一个包含若干参数（也称为*权重*或*系数*）的模型，如线性回归或支持向量机。RFE 首先会使用所有的特征训练模型，然后找出具有最小参数的特征（这代表该特征并不重要），并将它从特征集中移除。注意，在该方法中我们假设特征已经进行过标准化处理。

一个显而易见的问题是：应该保留多少特征？我们可以重复这个循环，直到只剩下最后一个特征。另一个选择是引入一种被称为交叉验证（CV）的方法。我们将在下一章中详细讨论交叉验证，这里只介绍总体思路。

假设有两种数据：要预测的目标向量和特征矩阵。第一步，将数据分为两组：训练集和测试集。第二步，使用训练集来训练模型。第三步，假装不知道测试集的目标向量（预测结果），将模型应用于测试集的特征，得到测试集的预测结果。最后，将测试集的预测结果与真实值进行比较，以评估模型性能。

可以通过交叉验证方法计算出 RFE 期间需要保留的最优特征数量。具体做法如下：在每次 RFE 迭代中，使用交叉验证来评估模型。如果交叉验证显示在减少一个特征后模型的性能有所改进，则继续下一次循环。如果交叉验证显示在丢弃某个特征后模型的性能变差，则将该特征重新加入到特征集，并选择这些特征作为最优特征。

在 scikit-learn 中，带有交叉验证的递归式特征消除是由 RFECV 类实现的，它包含许多重要参数。参数 `estimator` 决定训练模型的类型（如线性回归），参数 `step` 决定每次迭代丢弃的特征的数量或比例，参数 `scoring` 决定在做交叉验证时评估模型性能的方法。

延伸阅读

- 使用交叉验证进行递归式特征消除（*http://bit.ly/2Ftuffz*）

第 11 章

模型评估

11.0 简介

本章我们将学习如何评估算法生成的模型。在学习如何创建模型之前就讨论如何评估模型的性能，这看起来多少有点奇怪，但我们这样做是有充分的理由的。一个模型只有当其预测结果准确时才有实用价值，所以训练模型的目的并不仅仅是创建一个模型（这很简单），更要创建一个准确的模型（这很难）。因此，在开始学习各种模型算法之前，需要先了解如何评估生成的模型。

11.1 交叉验证模型

问题描述

评估算法生成的模型在实际应用中的表现。

解决方案

创建一个流水线，对数据进行预处理、训练模型，然后用交叉验证方法评估模型的性能。

```
# 加载库
from sklearn import datasets
from sklearn import metrics
from sklearn.model_selection import KFold, cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
```

```
# 加载手写数字的数据集
digits = datasets.load_digits()

# 创建特征矩阵
features = digits.data

# 创建目标向量
target = digits.target

# 创建标准化对象
standardizer = StandardScaler()

# 创建逻辑回归对象
logit = LogisticRegression()

# 创建包含数据标准化和逻辑回归的流水线
pipeline = make_pipeline(standardizer, logit)

# 创建 k 折交叉验证对象
kf = KFold(n_splits=10, shuffle=True, random_state=1)

# 执行 k 折交叉验证
cv_results = cross_val_score(pipeline, # 流水线
                             features, # 特征矩阵
                             target, # 目标向量
                             cv=kf, # 交叉验证方法
                             scoring="accuracy", # 损失函数
                             n_jobs=-1) # 使用所有的 CPU 核

# 计算得分的平均值
cv_results.mean()
0.96493171942892597
```

讨论

评估有监督学习模型的方法非常直观：首先训练一个模型，然后计算一些性能指标（准确率、均方误差等）来评估它的预测效果。然而这种方法从根本上说是有缺陷的。如果使用同一份数据来训练和评估模型，那么得到的结果可能并不是我们想要的。评估模型的目的并不是要评估其在训练集数据上的表现，而是要评估它在从未见过的数据上（如新客户、新犯罪、新图像）的表现。因此，评估方法应该帮助我们评估模型在新数据上做预测时的准确性。

一种可行的策略是保留一部分数据作为测试数据，这种方式被称为 Hold-out 验证（Hold-

out validation)。在 Hold-out 验证中，数据（特征值和目标向量）被分成训练集（*training set*）和测试集（*test set*）两部分。首先将测试集放在一边，假定这部分数据不存在。然后使用训练集训练模型，使用特征矩阵和目标向量来教模型做出最佳预测。最后，通过评估模型在测试集上的预测效果，来模拟该模型在从未见过的数据上的预测效果。不过这种方法有两大缺点：首先，模型的性能高度依赖于所选取的测试集；其次，模型在训练和评估时都没有充分利用所有可用数据。

有一种方法可以克服以上缺陷，叫作 *k* 折交叉验证（K-Fold Cross-Validation，KFCV）。在 *k* 折交叉验证中，数据被分为 *k* 份，训练模型时将 *k*-1 份数据组合起来作为训练集，剩下的那一份数据作为测试集。将上述过程重复 *k* 次，每次取一份不同的数据作为测试集，对模型在 *k* 次迭代中的得分取平均值作为总体得分。

在本节的解决方案中，数据被分为 10 份来做交叉验证，并将评估结果输出到 `cv_results` 中。

```
# 查看 10 份数据的得分
cv_results
array([ 0.97222222,  0.97777778,  0.95555556,  0.95      ,  0.95555556,
        0.98333333,  0.97777778,  0.96648045,  0.96089385,  0.94972067])
```

在使用 KFCV 时，需要考虑三点。首先，KFCV 假定每个样本都是独立于其他样本的（即数据是独立同分布的，Independent Identically Distributed，IID）。如果数据是独立同分布的，最好在数据分组前将其顺序打乱。在 scikit-learn 中，可以通过设置 `shuffle = True` 来打乱数据。

其次，当我们使用 KFCV 来评估分类器时，通常将每一类数据大致平均地分配到 *k* 组数据中（这被称为分层 *k* 折）。例如，如果目标向量中包含性别，并且 80%的样本是男性，则每组数据都包含 80%的男性和 20%的女性样本。在 scikit-learn 中，可以用 `StratifiedKFold` 替换 `KFold` 来使用分层 *k* 折交叉验证。

最后，使用 Hold-out 验证或交叉验证时，应该基于训练集对数据进行预处理，然后将这些预转换同时应用于训练集和测试集，这是非常重要的。例如，使用 `standardizer` 对数据进行标准化时，我们只需要计算训练集的平均值和方差，然后将这个预处理转换（使用 `transform` 方法）同时应用于训练集和测试集：

```
# 加载库
from sklearn.model_selection import train_test_split

# 创建训练集和测试集
features_train, features_test, target_train, target_test = train_test_
split(features, target, test_size=0.1, random_state=1)
```

```
# 使用训练集计算标准化参数
standardizer.fit(features_train)

# 将标准化操作应用到训练集和测试集
features_train_std = standardizer.transform(features_train)
features_test_std = standardizer.transform(features_test)
```

这样做是因为我们假定测试集是未知数据。如果使用训练集和测试集中的样本一起训练预处理器，测试集中的一些信息就会在训练过程中泄露。这个规则适用于任何预处理步骤，比如特征选择。

有了 scikit-learn 的流水线（pipeline），使用交叉验证就变得十分简单。首先创建一个预处理数据的流水线（如 `standardizer`），然后训练一个模型（`logit` 逻辑回归）：

```
# 创建一个流水线
pipeline = make_pipeline(standardizer, logit)
```

然后，只要对创建好的流水线执行 `cross_val_score` 方法，scikit-learn 就会帮我们完成所有工作。

```
# 执行 K 折交叉验证
cv_results = cross_val_score(pipeline, # 流水线
                             features, # 特征矩阵
                             target, # 目标向量
                             cv=kf, # 交叉验证方法
                             scoring="accuracy", # 损失函数
                             n_jobs=-1) # 使用所有的 CPU 核
```

`cross_val_score` 方法带有三个我们尚未讨论过的参数，这三个参数也很重要。 `cv` 参数用于指定使用什么样的交叉验证方法。*k* 折交叉验证是迄今为止最常见的方法，也有一些其他的方法，如留一交叉验证法（leave-one-out-cross-validation），该方法的折数 *k* 等于样本数。`scoring` 参数指定了衡量模型性能的标准，本章的其他节会讨论这些标准。最后，`n_jobs = -1` 告诉 scikit-learn 使用所有可用的 CPU 核来进行计算。例如，如果你的计算机有 4 个核（一般笔记本电脑中的内核数），那么 scikit-learn 将使用全部 4 个 CPU 核来加速操作。

延伸阅读

- 为什么每个统计学家都应该了解交叉验证（*http://bit.ly/2Fzhz6X*）
- 《交叉验证中的陷阱》（*http://bit.ly/2FzfIiw*）

11.2 创建一个基准回归模型

问题描述

需要一个简单基准回归模型，用于与算法生成的模型进行对比。

解决方案

使用 scikit-learn 的 DummyRegressor 创建一个简单的基准回归模型：

```
# 加载库
from sklearn.datasets import load_boston
from sklearn.dummy import DummyRegressor
from sklearn.model_selection import train_test_split

# 加载数据
boston = load_boston()

# 创建特征矩阵和目标向量
features, target = boston.data, boston.target

# 将数据分为测试集和训练集
features_train, features_test, target_train, target_test = train_test_split(
    features, target, random_state=0)

# 创建 DummyRegressor 对象
dummy = DummyRegressor(strategy='mean')

# 训练回归模型
dummy.fit(features_train, target_train)

# 计算 R 方得分
dummy.score(features_test, target_test)

-0.0011193592039553391
```

接下来训练自己的模型，计算性能得分，并与基准模型进行比较：

```
# 加载库
from sklearn.linear_model import LinearRegression

# 训练简单的线性回归模型
ols = LinearRegression()
ols.fit(features_train, target_train)

# 计算R方得分
```

```
ols.score(features_test, target_test)
0.63536207866746675
```

讨论

DummyRegressor 允许我们创建一个简单的模型，以此作为基准和实际的模型进行对比。我们通常用这种方法来模拟某个产品或系统中已有的原始预测系统。例如，一个产品将“所有新用户都会在第一个月消费 100 美元”这条规则硬编码到其预测系统中，而忽视了用户的所有特征。如果我们将这个假设编码到基准模型中，就能将机器学习方法的优势具体地展现出来。

DummyRegressor 使用 strategy 参数设置所使用的预测方法，可选的方法包括训练集的均值或者中位数。此外，如果将 strategy 设置成 constant 并使用 constant 参数，则模型对所有样本的预测结果都会是一个常数：

```
# 创建一个将所有样本预测为 20 的 DummyRegressor
clf = DummyRegressor(strategy='constant', constant=20)
clf.fit(features_train, target_train)

# 计算模型的得分
clf.score(features_test, target_test)

-0.065105020293257265
```

默认情况下，score 方法返回的是决定系数（R-squared，R^2 得分）：

$$R^2 = 1 - \frac{\Sigma_i(y_i - \hat{y}_i)^2}{\Sigma_i(y_i - \bar{y})^2}$$

其中，y_i 是目标样本的真实值，$\hat{y}_i$ 是预测值，$\bar{y}$ 是目标向量的均值。

R^2 越接近 1，就代表特征对目标向量的解释越好（即相关性越高）。

11.3 创建一个基准分类模型

问题描述

想要一个简单的基准分类模型与其他模型进行对比。

解决方案

使用 scikit-learn 的 DummyClassifier：

```
# 加载库
from sklearn.datasets import load_iris
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import train_test_split

# 加载数据
iris = load_iris()

# 创建特征矩阵和目标向量
features, target = iris.data, iris.target

# 将数据分为测试集和训练集
features_train, features_test, target_train, target_test = train_test_split(
features, target, random_state=0)

# 创建 DummyClassifier
dummy = DummyClassifier(strategy='uniform', random_state=1)

# 训练模型
dummy.fit(features_train, target_train)

# 计算模型的得分
dummy.score(features_test, target_test)

0.42105263157894735
```

通过比较基准分类模型和我们训练的分类模型的得分，可以看到训练的模型在性能方面的提升：

```
# 加载库
from sklearn.ensemble import RandomForestClassifier

# 创建分类模型
classifier = RandomForestClassifier()

# 训练模型
classifier.fit(features_train, target_train)

# 计算模型的得分
classifier.score(features_test, target_test)

0.94736842105263153
```

讨论

一种常用的衡量分类模型（也即分类器）性能的标准是将分类器的准确率与随机猜测的分类结果进行对比。scikit-learn 的 `DummyClassifier` 可以简化这个比较过程，它的 `strategy` 参数提供了一些生成预测值方法的选项，其中有两个非常有用。第一个是 `stratified` 方法，它使预测结果与训练集中数据类别的比例相同。例如，训练集中有 20%是女性，则 `DummyClassifier` 会有 20%的概率给出女性作为预测结果；第二个是 `uniform` 方法，它随机生成均匀的预测，如果样本中包含 20%的女性，80%的男性，那么 `uniform` 算法产生的预测结果中 50%为女性，50%为男性。

延伸阅读

scikit-learn 文档：DummyClassifier（*http://bit.ly/2Fr178G*）

11.4 评估二元分类器

问题描述

给定一个训练后的二元分类器，评估它的性能。

解决方案

使用 scikit-learn 的 `cross_val_score` 方法进行交叉验证，同时使用 `scoring` 参数来决定性能评估指标，可以在准确率、精确度、召回率和 F1 分数等多种指标中选择。

准确率（accuracy）是一种常见的性能指标，它表示被正确预测的样本数占参与预测的样本总数的比例，其计算公式如下：

$$准确率=\frac{TP+TN}{TP+TN+FP+FN}$$

其中：

- TP 是“真阳性”的数量，代表本身属于*正类*（如患有此类疾病、已购买此产品等），而且被正确预测的样本。
- TN 是“真阴性”的数量，代表本身属于*负类*（如未患有此类疾病、没有购买过该产品等），而且被正确预测的样本。
- FP 是“假阳性”的数量，也被称为 I 类错误，代表本来属于负类，却被错误预测为正类的样本。

- FN 是“假阴性”的数量，也称为 II 类错误，代表本来属于正类，却被错误预测为负类的样本。

我们可以通过设定 scoring="accuracy" 来使用 3 折（3 为默认的折数）交叉验证评估模型的准确率。

```
# 加载库
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification

# 生成特征矩阵和目标向量
X, y = make_classification(n_samples = 10000,
                           n_features = 3,
                           n_informative = 3,
                           n_redundant = 0,
                           n_classes = 2,
                           random_state = 1)

# 创建逻辑回归对象
logit = LogisticRegression()

# 使用准确率对模型进行交叉验证
cross_val_score(logit, X, y, scoring="accuracy")

array([ 0.95170966,  0.9580084 ,  0.95558223])
```

准确率的优点在于它有直观的解释：被正确预测的样本的百分比。然而，在现实世界中，样本数据的分布往往会有类别不均衡的问题（例如，99.9%的样本属于分类 1，只有 0.1% 属于分类 2）。当存在类别不均衡的情况时，使用准确率作为衡量标准会出现模型的准确率高但预测能力不足的情况。例如，我们想预测一种非常罕见的癌症（只有 0.1%的发病率）在人群中的分布情况。模型训练完以后，发现其准确率达到 95%。然而，99.9% 的人没有罹患这种癌症，也就是说，如果我们随便创建一个简易的模型“预测”根本没有人患这种癌症，这个模型的准确率也会比训练出的模型高 4.9%，但显然这没有任何意义，什么也预测不了。出于这个原因，我们通常会使用一些其他指标（如精确度、召回率和 F1 分数）来评估模型。

精确度（precision）是所有被预测为正类的样本中被正确预测的样本的百分比。我们可以把它看作一种衡量预测结果中的噪声的指标，即当样本被预测为正类时，预测正确的可能性有多大。具有高精确度的模型是悲观的，因为它们仅在非常确定时才会预测样本为正类。精确度可以用如下公式表示：

$$精确度 = \frac{TP}{TP+EP}$$

```
# 使用精确度对模型进行交叉验证
cross_val_score(logit, X, y, scoring="precision")

array([ 0.95252404,  0.96583282,  0.95558223])
```

召回率是“真阳性”样本占所有正类样本的比例。召回率衡量的是模型识别正类样本的能力。召回率高的模型是乐观的，因为它们比较容易将样本预测为正类：

$$召回率 = \frac{TP}{TP+FN}$$

```
# 使用召回率进行交叉验证
cross_val_score(logit, X, y, scoring="recall")

array([ 0.95080984,  0.94961008,  0.95558223])
```

如果这是你第一次听说精确度和召回率，可能需要花一点时间才能完全理解这些概念。这两个指标与准确率相比都有一个缺点，就是不那么直观。大多数情况下，我们希望在精确度和召回率之间达到某种平衡，而 F_1 分数（F_1 score）可以满足这种需求。 F1 分数是调和平均值（一种用于概率数据的平均数）：

$$F_1 = 2 \times \frac{精确度 \times 召回率}{精确度 + 召回率}$$

F_1 分数是衡量正类预测的正确程度（correctness）的指标，代表在被标记为正类的样本中，确实是正类的样本所占的比例。

```
# 使用 f1 分数进行交叉验证
cross_val_score(logit, X, y, scoring="f1")

array([ 0.95166617,  0.95765275,  0.95558223])
```

讨论

作为评估指标，准确率包含一些有价值的特性，而且它非常易于理解。然而，好的衡量标准往往还要在精确度和召回率之间保持某种平衡，即模型要在乐观和悲观之间保持平衡。 分数代表了召回率与精确度之间的一种平衡，两者的相对贡献是相等的。

除了使用 `cross_val_score` 方法，如果已经得到样本的真实值和预测值，我们还可以直

接计算出准确率和召回率等指标：

```
# 加载库
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# 创建训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.1,
                                                    random_state=1)

# 对测试集进行预测
y_hat = logit.fit(X_train, y_train).predict(X_test)

# 计算准确率
accuracy_score(y_test, y_hat)

0.94699999999999995
```

延伸阅读

- 准确率的悖论（*http://bit.ly/2FxTpK0*）

11.5 评估二元分类器的阈值

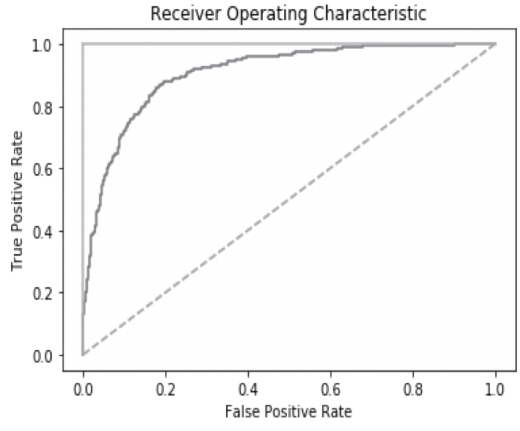
问题描述

希望评估二元分类器的性能，并对多个可能的阈值进行评估。

解决方案

受试者工作特征（Receiving Operating Characteristic，ROC）曲线是评估二元分类器质量的常用方法。ROC 曲线会对每一个概率阈值（即用来区分样本属于正类或负类的概率值）比较其真阳性和假阳性的比例。通过绘制 ROC 曲线，我们可以观察模型的性能。一个正确预测了每个样本的分类器，其 ROC 曲线看起来就像 197 页的图中的实心浅灰线，直接从原点到达顶部。而一个随机预测的分类器，其 ROC 曲线将显示为对角线。模型的性能越好，ROC 曲线就越接近实心浅灰线。在 scikit-learn 中，我们可以使用 `roc_curve` 来计算每个阈值下的真阳性率（True Postive Rate）和假阳性率（False Positive Rate），然后用图绘制出来：

```
    # 加载库
    import matplotlib.pyplot as plt
    from sklearn.datasets import make_classification
    from sklearn.linear_model import LogisticRegression
    from sklearn.metrics import roc_curve, roc_auc_score
    from sklearn.model_selection import train_test_split

    # 创建特征矩阵和目标向量
    features, target = make_classification(n_samples=10000,
                                           n_features=10,
                                           n_classes=2,
                                           n_informative=3,
                                           random_state=3)
    # 将样本划分为测试集和训练集
    features_train, features_test, target_train, target_test = train_test_split(
        features, target, test_size=0.1, random_state=1)

    # 创建分类器
    logit = LogisticRegression()

    # 训练模型
    logit.fit(features_train, target_train)

    # 获取预测的概率
    target_probabilities = logit.predict_proba(features_test)[:,1]

    # 计算真阳性和假阳性的概率
    false_positive_rate, true_positive_rate, threshold = roc_curve(target_test,
                                                                   target_
probabilities)
    # 画出 ROC 曲线
    plt.title("Receiver Operating Characteristic")
    plt.plot(false_positive_rate, true_positive_rate)
    plt.plot([0, 1], ls="--")
    plt.plot([0, 0], [1, 0] , c=".7"), plt.plot([1, 1] , c=".7")
    plt.ylabel("True Positive Rate")
    plt.xlabel("False Positive Rate")
    plt.show()
```



讨论

到目前为止，我们只是基于模型的预测值来检验模型。然而，在许多学习算法中预测值是基于概率估计的。也就是说，每个样本都有一定的概率归属于不同的分类。我们可以使用 predict_proba 查看第一个样本的预测概率：

```
# 获取预测的概率
logit.predict_proba(features_test)[0:1]

array([[ 0.8688938, 0.1311062]])
```

可以使用 classes_ 来查看分类：

```
logit.classes_
 array([0, 1])
```

在这个例子中，第一个样本有约 87% 的概率属于负类（0），有约 13% 的概率属于正类（1）。默认情况下，scikit-learn 会将样本预测为概率大于 50%（即阈值）的分类。然而，通常我们会由于某些原因显式地调整阈值而不是使用这个中间值。例如，如果假阳性对业务的影响很大，那么我们会更倾向于选用一个较大的概率阈值。这样做虽然会导致有些真阳性的样本被误判，但是如果一个样本被预测为正类，我们就确信模型的预测基本上是正确的。这种折衷表现为真阳性率（TPR）和假阳性率（FPR）。真阳性率是被正确预测的正类样本数与所有被预测为正类的样本数之比：

$$\text{真阳性率（TPR）}=\frac{\text{真阳性的样本数量}}{\text{真阳性的样本数量}+\text{假阴性的样本数量}}$$

假阳性率是被错误预测为负类的样本数和所有被预测为负类的样本数之比：

$$\text{假阳性率（FPR）}=\frac{\text{假阳性的样本数量}}{\text{假阳性的样本数量}+\text{真阴性的样本数量}}$$

ROC 曲线表示每个概率阈值下相应的 TPR 和 FPR 值。例如，在本节的解决方案中，在约 50% 的概率阈值下，TPR 为 0.81，FPR 为 0.15 ：

```
print("Threshold:", threshold[116])
print("True Positive Rate:", true_positive_rate[116])
print("False Positive Rate:", false_positive_rate[116])

Threshold: 0.528224777887
True Positive Rate: 0.810204081633
False Positive Rate: 0.154901960784
```

然而，如果将阈值提高到 80%左右（即提高模型将样本预测为正类所需的确信度），则 TPR 和 FPR 的值都会显著下降 ：

```
print("Threshold:", threshold[45])
print("True Positive Rate:", true_positive_rate[45])
print("False Positive Rate:", false_positive_rate[45])

Threshold: 0.808019566563
True Positive Rate: 0.563265306122
False Positive Rate: 0.0470588235294
```

这是因为我们对样本被预测为正类的要求提高了，模型无法识别出一些正类样本（TPR 较小），而且被错误预测为正类的负类样本也减少了（FPR 较小）。

除了能够可视化 TPR 和 FPR 之间的关系，ROC 曲线还可以用作模型的通用度量标准。模型越好，曲线越高，曲线下的面积也就越大。出于这个原因，通常会计算 ROC 曲线下方的面积（AUCROC）来判断在所有可能的阈值下模型的总体性能水平。AUCROC 的值越接近 1，模型的性能就越好。在 scikit-learn 中，可以使用 `roc_auc_score` 来计算 AUCROC ：

```
# 计算 ROC 曲线下方的面积
roc_auc_score(target_test, target_probabilities)

0.90733893557422962
```

延伸阅读

- 《Python 和 R 中的 ROC 曲线》（*http://bit.ly/2FuqoyV*）
- 《ROC 曲线下方的面积》（*http://bit.ly/2FxTrl6*）

11.6 评估多元分类器

问题描述

评估一个能预测三个或更多分类的分类器性能。

解决方案

使用能够处理两个以上分类的评估指标进行交叉验证：

```
# 加载库
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification

# 生成特征矩阵和目标向量
features, target = make_classification(n_samples = 10000,
                            n_features = 3,
                            n_informative = 3,
                            n_redundant = 0,
                            n_classes = 3,
                            random_state = 1)

# 创建逻辑回归
logit = LogisticRegression()

# 使用准确率作为指标进行交叉验证
cross_val_score(logit, features, target, scoring='accuracy')

array([ 0.83653269,  0.8259826 ,  0.81308131])
```

讨论

如果分类数据均衡（比如，每个分类的样本数大致相等），准确率是评估模型性能的一个简单且可解释的指标。准确率是被正确预测的样本数与样本总数之比，并且在二元分类器和多元分类器中的效果一样好。但是，当分类数据不均衡时（这种情况很常见），应该使用其他评估指标。

scikit-learn 许多内置的指标都是用于评估二元分类器的，但是其中的一部分指标也可以应用在多元分类器上。精确度、召回率和 F_1 分数是在前面几节已经详细介绍过的有用指标。虽然它们最初都是为二元分类器设计的，但我们可以将数据看作一组二元分类数据的集合，使用上述指标来评估多元分类器。这就能对每一个分类应用二元分类器的指标，就好像它是数据中的唯一分类一样，然后对所有分类的得分求平均值，得到总分数：

```
# 使用 macro-F1 分数对模型进行交叉验证
cross_val_score(logit, features, target, scoring='f1_macro')

array([ 0.83613125,  0.82562258,  0.81293539])
```

在上面的代码中，_macro 指用来求各分类评估分数的平均值的方法：

`macro`

计算每个分类的得分，然后取加权平均值，每个分类的权值相同。

`weighted`

计算每个分类的得分，然后取加权平均值，权值为每个分类的样本数占总样本数的比例。

`micro`

计算每个样本分类组合的得分，然后取平均值。

11.7 分类器性能的可视化

问题描述

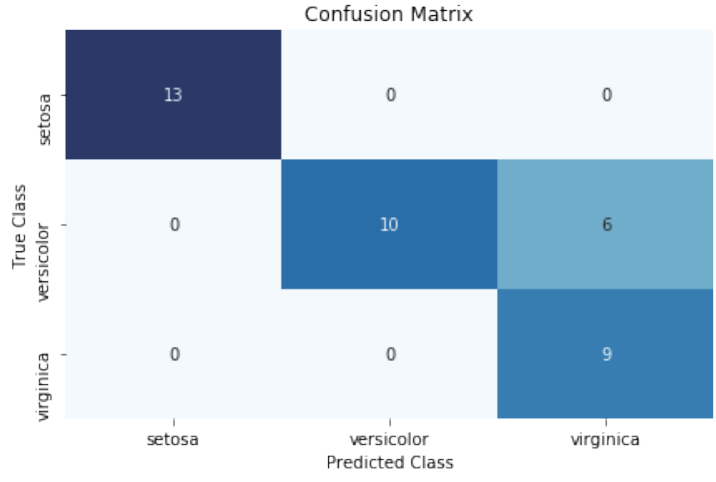
给定测试集数据的预测分类和真实分类，希望能够可视化地比较不同模型的性能。

解决方案

使用混淆矩阵（Confusion Matrix, 如 202 页的图所示）来比较预测分类（Predicted Class）和真实分类（True Class）：

```
# 加载库
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import confusion_matrix
import pandas as pd

# 加载数据
iris = datasets.load_iris()

# 创建特征矩阵
features = iris.data

# 创建目标向量
target = iris.target

# 创建目标分类的名称列表
class_names = iris.target_names

# 创建训练集和测试集
features_train, features_test, target_train, target_test = train_test_split(
    features, target, random_state=1)

# 创建逻辑回归分类模型
classifier = LogisticRegression()

# 训练模型并做出预测
target_predicted = classifier.fit(features_train,
    target_train).predict(features_test)

# 创建混淆矩阵
matrix = confusion_matrix(target_test, target_predicted)

# 创建一个 pandas DataFrame
dataframe = pd.DataFrame(matrix, index=class_names, columns=class_names)

# 绘制热力图
sns.heatmap(dataframe, annot=True, cbar=None, cmap="Blues")
plt.title("Confusion Matrix"), plt.tight_layout()
plt.ylabel("True Class"), plt.xlabel("Predicted Class")
plt.show()
```



讨论

混淆矩阵是将分类器性能可视化的简单有效的方式。它的一个主要的优点就是很容易解释。矩阵（通常以热力图形式呈现）的每一列表示样本的预测分类，而每一行表示样本的真实分类。最终每个单元格都是预测分类和真实分类的一种可能的组合。或许用一个例子能清楚地解释。在本节的解决方案中，左上角的单元格是被预测为 *Iris setosa* 分类的样本数（由列表示），其真实分类也是 *Iris setosa* 分类的样本数（由行表示）。这说明我们的模型准确地预测了所有分类为 *Iris setosa* 的样本。然而，该模型在预测 *Iris virginica* 分类时表现不佳。右下角的单元格表明，该模型成功地预测了 9 个真实分类为 *Iris virginica* 的样本，但是（右侧中间的单元格）却将 6 个真实分类为 *Iris versicolor* 的样本预测为 *Iris virginica*。

关于混淆矩阵有三点值得注意。首先，一个完美的模型，其混淆矩阵应该只有对角线上才有值，而其他位置应该全为零；而一个糟糕的模型，其混淆矩阵看起来像就是将样本均匀分配到了各个单元格上。其次，混淆矩阵不仅可以显示模型将哪些样本的分类预测错了，还可以显示它是怎么分错的，即误分类的模式。例如，本节的解决方案中模型能轻松区分 *Iris virginica* 和 *Iris setosa* 的样本，却难以区分 *Iris virginica* 和 *Iris versicolor* 的样本。最后，对于任意数量的分类，混淆矩阵都适用（只是如果数据中有 100 万个分类，混淆矩阵可能就不那么直观了）。

延伸阅读

- 混淆矩阵（*http://bit.ly/2FuGKaP*）
- scikit-learn 文档：混淆矩阵（*http://bit.ly/2DmnICk*）

11.8 评估回归模型

问题描述

评估一个回归模型的性能。

解决方案

使用均方误差（Mean Squared Error，MSE）：

```
# 加载库
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression

# 生成特征矩阵和目标向量
features, target = make_regression(n_samples = 100,
                                   n_features = 3,
                                   n_informative = 3,
                                   n_targets = 1,
                                   noise = 50,
                                   coef = False,
                                   random_state = 1)

# 创建 LinerRegression 对象
ols = LinearRegression()

# 使用 MSE 对线性回归做交叉验证
cross_val_score(ols, features, target, scoring='neg_mean_squared_error')

array([-1718.22817783, -3103.4124284 , -1377.17858823])
```

另一个常用的评估回归模型的指标是决定系数（coefficient of determination，即得分）：

```
# 使用决定系数对线性回归进行交叉验证
cross_val_score(ols, features, target, scoring='r2')

array([ 0.87804558,  0.76395862,  0.89154377])
```

讨论

MSE 是回归模型最常用的评估指标之一。MSE 的计算公式如下：

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(\widehat{y_i} - y_i\right)^2$$

其中，n 是样本的数量，y_i 是样本 i 的真实值，$\widehat{y_i}$ 是模型对 y_i 的预测值，MSE 是预测值和真实值距离的平方和。MSE 的值越大，总平方误差就越大，模型性能越差。平方误差项在数学上有许多好的特性，比如所有误差值都为正数，但是有一个特性经常被忽略，即在误差的总绝对值相同的情况下，取平方值以后会给值较大的误差项施加更多的惩罚。例如，有两个模型 A 和 B，每个模型都有两个样本：

- 模型 A 中两个样本的误差分别是 0 和 10，它的 MSE 为 $0^2+10^2=100$。
- 模型 B 中两个样本的误差分别是 5 和 5，它的 MSE 为 $5^2+5^2=50$。

两个模型的总误差值相等，都是 10；然而，MSE 会认为模型 A（MSE = 100）的预测效果比模型 B（MSE = 50）的差。在实践中并不能像这样区分模型的好坏（只不过我们在理论上可以这么认为），并且 MSE 作为评估指标的效果很好。

有一点很重要：默认情况下，对于 scikit-learn 的 `scoring` 参数而言，`scoring` 值较大的模型优于 `scoring` 值较小的模型。然而对 MSE 来说，刚好是相反的，MSE 的值较大意味着模型性能较差。因此，在 scikit-learn 中使用的是 `neg_mean_squared_error` 参数，它是 MSE 的相反数。

另一个常见的评估回归模型的指标是决定系数（R^2 得分），它代表目标向量的变化中有多少能通过模型进行解释：

$$R^2 = 1 - \frac{\Sigma_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2}{\Sigma_{i=1}^{n}\left(y_i - \bar{y}\right)^2}$$

其中，y_i 是样本 i 的真实值，$\widehat{y_i}$ 是样本 y_i 的预测值，$\bar{y}$ 是目标向量的平均值。R^2 得分越接近 1，代表模型性能越好。

延伸阅读

- 均方误差（*http://bit.ly/2HgALnc*）
- 决定系数（*http://bit.ly/2HjW7Qn*）

11.9 评估聚类模型

问题描述

使用无监督学习对数据进行聚类，现在希望评估聚类模型的性能。

解决方案

对该问题的简单回答是，我们可能无法评估聚类模型的性能，至少不能按预想的方式评估。

一种可选的方式是使用轮廓系数（silhouette coefficient）。轮廓系数可以用来衡量聚类的质量：

```
import numpy as np
from sklearn.metrics import silhouette_score
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# 生成特征矩阵
features, _ = make_blobs(n_samples = 1000,
                         n_features = 10,
                         centers = 2,
                         cluster_std = 0.5,
                         shuffle = True,
                         random_state = 1)

# 使用 KMeans 方法对数据聚类，以预测其分类
model = KMeans(n_clusters=2, random_state=1).fit(features)

# 获取预测的分类
target_predicted = model.labels_

# 评估模型
silhouette_score(features, target_predicted)

0.89162655640721422
```

讨论

评估有监督学习模型时，要将预测值（比如，分类值或量化值）与目标向量中对应的真实值进行比较。但是，使用聚类方法最常见的原因是数据集没有目标向量。许多评估聚类模型的指标都要求有目标向量，但是当有可用的目标向量时，使用无监督学习（如聚类）

方法训练模型可能会带来一些不必要的麻烦。

如果没有目标向量，就无法评估预测值与真实值之间的差距，不过聚类（cluster）本身的特性仍然是可以评估的。直观地讲，“好”的聚类中同类别样本间的距离非常小（即稠密聚类），不同类别的样本之间距离非常大（即分离得很彻底）。轮廓系数可以用一个值同时评估这两种特性。第 i 个样本的轮廓系数的计算公式为：

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)}$$

其中，s_i 是样本 i 的轮廓系数，a_i 是样本 i 与同类的所有样本间的平均距离，b_i 是样本 i 与来自不同分类的最近聚类的所有样本间的平均距离。`silhouette_score` 返回的值是所有样本的平均轮廓系数。轮廓系数的值介于 −1 和 1 之间，其中 1 表示内部密集、分离彻底（指与其他聚类的间隔较远）的聚类。

延伸阅读

- scikit-learn 文档：silhouette_score（*http://bit.ly/2BEVQV5*）

11.10 创建自定义评估指标

问题描述

使用自定义指标来评估模型。

解决方案

将评估指标实现为一个函数，并使用 scikit-learn 的 `make_scorer` 将其转换为评分函数：

```
# 加载库
from sklearn.metrics import make_scorer, r2_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
from sklearn.datasets import make_regression

# 创建特征矩阵和目标向量
features, target = make_regression(n_samples = 100,
                                   n_features = 3,
                                   random_state = 1)
```

```
# 创建训练集和测试集
features_train, features_test, target_train, target_test = train_test_split(
     features, target, test_size=0.10, random_state=1)

# 创建自定义指标函数
def custom_metric(target_test, target_predicted):
    # 计算R方得分
    r2 = r2_score(target_test, target_predicted)
    # 返回R方得分
    return r2

# 创建评分函数（评分器），并且定义分数越高代表模型越好
score = make_scorer(custom_metric, greater_is_better=True)

# 创建岭回归（Ridge Regression）对象
classifier = Ridge()

# 训练岭回归模型
model = classifier.fit(features_train, target_train)

# 应用自定义评分器
score(model, features_test, target_test)

0.99979061028820582
```

讨论

虽然 scikit-learn 有许多用于评估模型性能的内置指标，但定义我们自己的指标仍然很有用。 使用 scikit-learn 的 `make_scorer` 函数，我们能很轻松地创建自定义指标函数。首先，定义一个函数，它接受两个参数——真实的目标向量和预测值，并返回一个分数。然后，使用 `make_scorer` 创建一个评分器对象，并指定较高的分数代表模型性能较好或者较差（使用 `greater_is_better` 参数）。

本例的解决方案中自定义指标 `custom_metric` 是一个简单的示例，它只是对内置的指标做了封装，用以计算得分。在实际情况中，可以使用任何自定义指标替换 `custom_metric` 函数。不过，将结果与 scikit-learn 内置的 `r2_score` 方法进行比较，我们可以确定这个计算得分的自定义指标能正常工作：

```
# 对测试集进行预测
target_predicted = model.predict(features_test)

# 计算R方得分
r2_score(target_test, target_predicted)

0.99979061028820582
```

延伸阅读

- scikit-learn 文档：make_scorer（*http://bit.ly/2FwMm4m*）

11.11 可视化训练集规模的影响

问题描述

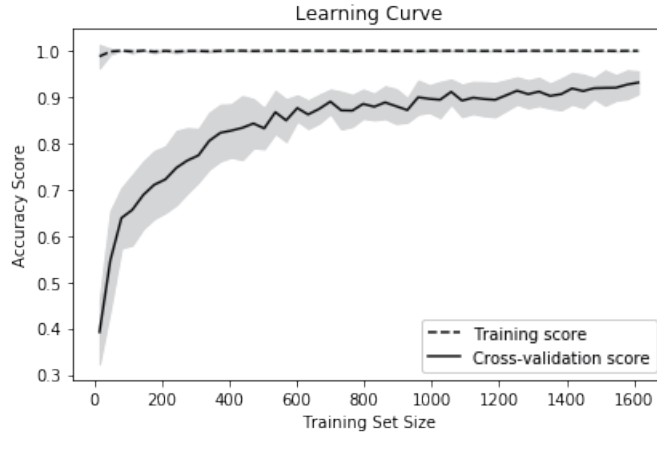
评估训练集中观察值的数量对某个指标的影响（如准确率、F_1 分数等）。

解决方案

画出学习曲线（Learning Curve，如下页图所示）：

```
# 加载库
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import learning_curve

# 加载数据
digits = load_digits()

# 创建特征矩阵和目标向量
features, target = digits.data, digits.target

# 使用交叉验证为不同规模的训练集计算训练和测试得分
train_sizes, train_scores, test_scores = learning_curve(# 分类器
                                                        RandomForestClassifier(),
                                                        # 特征矩阵
                                                        features,
                                                        # 目标向量
                                                        target,
                                                        # 交叉验证的折数
                                                        cv=10,
                                                        # 性能指标
                                                        scoring='accuracy',
                                                        # 使用所有 CPU 核
                                                        n_jobs=-1,
                                                        # 50 个训练集的规模
                                                        train_sizes=np.linspace(
```

```
                                                    0.01,
                                                    1.0,
                                                    50))
# 计算训练集得分的平均值和标准差
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)

# 计算测试集得分的平均值和标准差
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

# 画线
plt.plot(train_sizes, train_mean, '--', color="#111111", label="Training score")

plt.plot(train_sizes, test_mean, color="#111111", label="Cross-validation score")


# 画带状图
plt.fill_between(train_sizes, train_mean - train_std,
                 train_mean + train_std, color="#DDDDDD")
plt.fill_between(train_sizes, test_mean - test_std,
                 test_mean + test_std, color="#DDDDDD")

# 创建图
plt.title("Learning Curve")
plt.xlabel("Training Set Size"), plt.ylabel("Accuracy Score"),
plt.legend(loc="best")
plt.tight_layout()
plt.show()
```



讨论

学习曲线将模型在训练集上和交叉验证时的性能（例如准确率、召回率）以及与训练集样本数量之间的关系可视化地表达出来了。这种方法常常被用来判断增加训练集数据的规模能否提升模型的性能。

在解决方案中，我们绘制了随机森林分类器在 50 种不同规模（从样本总量的 1% 到 100%）的训练集上的准确率。模型交叉验证的准确率提升表明，样本数量越大，模型很可能获益越多（尽管在实践中获取更多数据不一定可行）。

延伸阅读

- scikit-learn 文档：学习曲线（*http://bit.ly/2FwjBVe*）

11.12 生成对评估指标的报告

问题

要快速描述一个分类器的性能。

解决方案

使用 scikit-learn 的 `classification_report` 函数：

```
# 加载库
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# 加载数据
iris = datasets.load_iris()

# 创建特征矩阵
features = iris.data

# 创建目标向量
target = iris.target

# 创建目标分类名的列表
class_names = iris.target_names
```

```
# 创建训练集和测试集
features_train, features_test, target_train, target_test = train_test_split(
    features, target, random_state=1)

# 创建逻辑回归对象
classifier = LogisticRegression()

# 训练模型并做出预测
model = classifier.fit(features_train, target_train)
target_predicted = model.predict(features_test)

# 生成分类器的性能报告
print(classification_report(target_test,
                            target_predicted,
                            target_names=class_names))
```

```
             precision    recall  f1-score   support

     setosa       1.00      1.00      1.00        13
 versicolor       1.00      0.62      0.77        16
  virginica       0.60      1.00      0.75         9

avg / total       0.91      0.84      0.84        38
```

讨论

`classification_report` 提供了快速查看一些常见模型评估指标的方法，这些指标包括精确度、召回率和 F_1 分数（本章前面有介绍）。输出中的 `support` 列是指每个分类中的样本数量。

延伸阅读

- 精确度和召回率（*https://en.wikipedia.org/wiki/Precision_and_recall*）

11.13 可视化超参数值的效果

问题描述

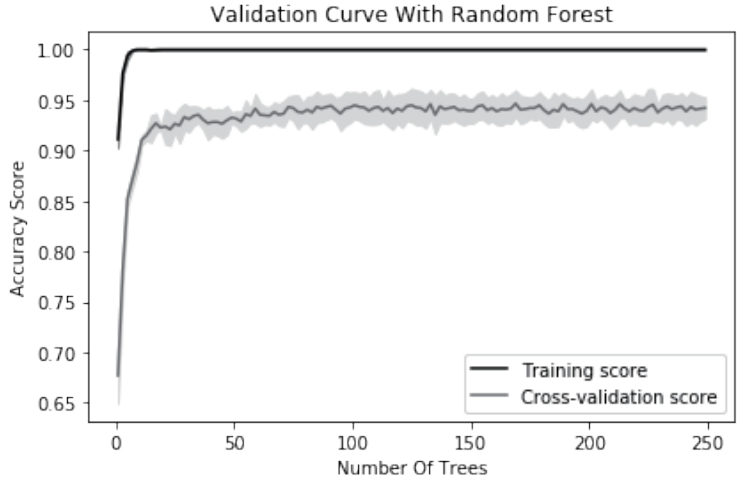
希望了解超参数值的变化对模型性能的影响。

解决方案

画出验证曲线（见下页图）：

```
# 加载库
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_digits
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import validation_curve

# 加载数据
digits = load_digits()

# 创建特征矩阵和目标向量
features, target = digits.data, digits.target

# 创建参数的变化范围
param_range = np.arange(1, 250, 2)

# 对区间内的参数值分别计算模型在训练集和测试集上的准确率
train_scores, test_scores = validation_curve(
    # 分类器
    RandomForestClassifier(),
    # 特征矩阵
    features,
    # 目标向量
    target,
    # 要查看的超参数
    param_name="n_estimators",
    # 超参数值的范围
    param_range=param_range,
    # 交叉验证的折数
    cv=3,
    # 性能指标
    scoring="accuracy",
    # 使用所有 CPU 核
    n_jobs=-1)

# 计算模型在训练集上得分的平均值和标准差
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)

# 计算模型在测试集上得分的平均值和标准差
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
```

```
    # 画出模型在训练集和测试集上的准确率的平均值
    plt.plot(param_range, train_mean, label="Training score", color="black")
    plt.plot(param_range, test_mean, label="Cross-validation score",
color="dimgrey")

    # 画出模型在训练集和测试集上的准确率带状图
    plt.fill_between(param_range, train_mean - train_std,
                     train_mean + train_std, color="gray")
    plt.fill_between(param_range, test_mean - test_std,
                     test_mean + test_std, color="gainsboro")

    # 画图
    plt.title("Validation Curve With Random Forest")
    plt.xlabel("Number Of Trees")
    plt.ylabel("Accuracy Score")
    plt.tight_layout()
    plt.legend(loc="best")
    plt.show()
```



讨论

大多数算法（包括本书所讲的许多算法）都包含超参数，而且必须在模型训练开始之前就选择好。例如，随机森林分类器创建决策树的“森林”，每个决策树对样本的预测分类投票。随机森林分类器中有一个超参数是森林中树的数量（Number of Trees）。大多数情况下，超参数的值是在模型选择阶段（参见第 12 章）确定的。但是，将模型性能与超参数值之间的关系画出来偶尔也很有用。上述解决方案中绘制的图展示了随着树的数

量增加，一个随机森林分类器在训练集上和交叉验证时的准确率的变化。当树的数量较少时，训练集和交叉验证的准确率分数都很低，表明该模型欠拟合。当树的数量增加到250，两个准确率分数 (Accuracy Score) 都没什么变化，表明使用更多计算资源训练大规模随机森林可能并不能带来很高价值。

在 scikit-learn 中，我们可以使用 `validation_curve` 计算验证曲线，它包含 3 个重要参数：

- `param_name` 是需要变化的超参数的名字。
- `param_range` 是超参数取值的区间。
- `scoring` 是模型的评估指标。

延伸阅读

- scikit-learn 文档：验证曲线（*http://bit.ly/2FuwYFG*）

第 12 章

模型选择

12.0 简介

在机器学习中，我们通过最小化某个损失函数的值来训练算法以学习一个模型的参数。此外，许多学习算法（如支持向量分类器和随机森林）还有一些超参数，这些超参数必须在学习过程之外定义。例如，随机森林是决策树的集合（因此称其*森林*）；但是，算法不会学习森林中决策树的数量，这个值必须在训练之前设定。这个过程通常被称为*超参数调优*（*hyperparameter tuning*）、*超参数优化*（*hyperparameter optimization*）或者*模型选择*（*model selection*）。除了超参数，你可能也想尝试多种学习算法（例如，尝试支持向量分类器和随机森林，以确定哪一种学习方法效果更好）。

这一领域的术语使用存在很大差异，在本书中，我们将选择最佳学习算法及选择最佳超参数的过程都称为模型选择。原因很简单：假设我们要训练具有 10 个候选超参数的支持向量分类器和具有 10 个候选超参数的随机森林分类器，实际上是要从 20 个候选模型中选出一个最佳模型。本章将介绍如何从候选模型中高效地选出最佳模型。

本章会谈到一些特定的超参数，例如 C（正则化强度的倒数）。如果你不知道这些参数的含义，请不要担心，后面的章节会介绍它们。你只需将超参数视为在训练学习算法前必须选择的参数即可。

12.1 使用穷举搜索选择最佳模型

问题描述

通过搜索一系列的超参数来选择最佳模型。

解决方案

使用 sciket-learn 的 GridSearchCV：

```
# 加载库
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建逻辑回归对象
logistic = linear_model.LogisticRegression()

# 创建正则化惩罚的候选超参数区间
penalty = ['l1', 'l2']

# 创建正则化候选超参数区间
C = np.logspace(0, 4, 10)

# 创建候选超参数的字典
hyperparameters = dict(C=C, penalty=penalty)

# 创建网格搜索对象
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, verbose=0)

# 训练网格搜索
best_model = gridsearch.fit(features, target)
```

讨论

GridSearchCV 是一种使用交叉验证进行模型选择的暴力方法。具体来说，就是用户为一个或多个超参数定义候选值的集合，然后 GridSearchCV 使用其中的每个值或值的组合来训练模型。性能得分最高的模型被选为最佳模型。例如，本节的解决方案使用逻辑回归作为学习算法，它包含两个超参数：C 和正则化惩罚（regularization penalty）。如果你不知道 C 和正则化惩罚代表什么意思，不要担心，我们将在接下来的几章中介绍它们。目前你只需知道 C 和正则化惩罚可以选用一系列值，而这些值必须在训练之前指定。解决方案中首先定义了 C 的 10 个可能的值：

```
np.logspace(0, 4, 10)
```

```
array([  1.00000000e+00,   2.78255940e+00,   7.74263683e+00,
         2.15443469e+01,   5.99484250e+01,   1.66810054e+02,
         4.64158883e+02,   1.29154967e+03,   3.59381366e+03,
         1.00000000e+04])
```

然后，为正则化惩罚也定义了两个可能的值：`['l1', 'l2']`。GridSearchCV 使用 C 和正则化惩罚的所有组合来训练模型，并使用 *k* 折交叉验证对模型进行评估。我们的解决方案中有 10 个可能的 C 值，2 个可能的正则化惩罚值，并采用 5 折交叉验证，这样的话一共有 10 × 2 × 5 = 100 个候选模型，要从其中选出最佳模型。

一旦 GridSearchCV 完成训练，就可以通过下面的方法查看最佳模型的超参数（最佳超参数）：

```
# 查看最佳超参数
print('Best Penalty:', best_model.best_estimator_.get_params()['penalty'])
print('Best C:', best_model.best_estimator_.get_params()['C'])
```

```
Best Penalty: l1
Best C: 7.74263682681
```

默认情况下，在找到最佳超参数后，GridSearchCV 将使用最佳超参数和整个数据集重新训练模型（而不是保留部分数据用于交叉验证）。我们可以像使用其他 scikit-learn 模型一样使用此模型来进行预测：

```
# 预测目标向量
best_model.predict(features)
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

GridSearchCV 的 `verbose` 参数值得注意，虽然该参数一般不需要设置，但在长搜索过程中可以设置该参数来获取一些过程进展信息。`verbose` 参数决定了搜索期间输出的消息量，`0` 表示没有输出，`1` 到 `3` 代表输出信息，数字越大表示输出的细节越多。

延伸阅读

- scikit-learn 文档：GridSearchCV（*http://bit.ly/2Fuctc4*）

12.2 使用随机搜索选择最佳模型

问题描述

使用一种比穷举搜索更节省计算资源的方法来选择最佳模型。

解决方案

使用 scikit-learn 的 RandomizedSearchCV：

```
# 加载库
from scipy.stats import uniform
from sklearn import linear_model, datasets
from sklearn.model_selection import RandomizedSearchCV

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建逻辑回归对象
logistic = linear_model.LogisticRegression()

# 创建正则化惩罚的候选超参数区间
penalty = ['l1', 'l2']

# 创建正则化候选超参数的分布
C = uniform(loc=0, scale=4)

# 创建超参数字典
hyperparameters = dict(C=C, penalty=penalty)

# 创建随机搜索对象
randomizedsearch = RandomizedSearchCV(
    logistic, hyperparameters, random_state=1, n_iter=100, cv=5, verbose=0,
    n_jobs=-1)

# 训练随机搜索
best_model = randomizedsearch.fit(features, target)
```

讨论

在 12.1 节中，我们在一组用户定义的超参数值上使用 GridSearchCV，并根据得分搜索最佳模型。比 GridSearchCV 的暴力搜索更有效的方法是，在用户提供的参数分

布（如正态分布、均匀分布）上选取特定数量的超参数随机组合。scikit-learn 使用 RandomizedSearchCV 实现这种随机搜索。

使用 RandomizedSearchCV 时，如果指定分布，那么 scikit-learn 将从该分布中对超参数进行无放回的随机采样。举一个例子，我们从区间 (0, 4) 的均匀分布中随机抽取 10 个样本值：

```
# 定义区间 (0, 4) 上的均匀分布，并从中抽取 10 个样本值
uniform(loc=0, scale=4).rvs(10)

array([ 1.97118721,  1.34786898,  3.79867719,  1.15999967,  3.33485021,
        1.20394179,  0.88117832,  0.47855329,  0.48521699,  1.5479854 ])
```

或者，如果指定一个值列表，例如两个正则化惩罚超参数值，`['l1','l2']`，RandomizedSearchCV 将对列表进行有放回的随机抽样。

和 GridSearchCV 一样，我们可以查看最佳模型的超参数值：

```
# 查看最佳超参数
print('Best Penalty:', best_model.best_estimator_.get_params()['penalty'])
print('Best C:', best_model.best_estimator_.get_params()['C'])

Best Penalty: l1
Best C: 1.66808801881
```

就像 GridSearchCV 一样，在搜索结束后 RandomizedSearchCV 使用整个数据集和最佳超参数训练一个新模型。该模型和 scikit-learn 中的其他模型一样可以用来做预测，例如像下面这样做预测：

```
# 预测目标向量
best_model.predict(features)

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

对超参数组合的采样数（即候选模型的数量）由参数 n_iter（迭代次数）指定。

延伸阅读

- scikit-learn 文档：RandomizedSearchCV（*http://bit.ly/2B7p1zT*）

- 《使用随机搜索优化超参数》（*http://bit.ly/2FrUinf*）

12.3 从多种学习算法中选择最佳模型

问题描述

通过搜索一系列学习算法及其超参数来选择最佳模型。

解决方案

为候选的学习算法及其超参数创建一个字典：

```
# 加载库
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline

# 设置随机数种子
np.random.seed(0)

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建流水线
pipe = Pipeline([("classifier", RandomForestClassifier())])

# 创建候选学习算法及超参数的字典
search_space = [{"classifier": [LogisticRegression()],
                 "classifier__penalty": ['l1', 'l2'],
                 "classifier__C": np.logspace(0, 4, 10)},
                {"classifier": [RandomForestClassifier()],
                 "classifier__n_estimators": [10, 100, 1000],
                 "classifier__max_features": [1, 2, 3]}]

# 创建 GridSearchCV 对象
gridsearch = GridSearchCV(pipe, search_space, cv=5, verbose=0)

# 执行网格搜索
best_model = gridsearch.fit(features, target)
```

讨论

在前两节中，我们通过搜索学习算法的候选超参数值找到了最佳模型。但是，如果不知道应该使用哪种学习算法该怎么办呢？最新版本的 scikit-learn 允许将学习算法作为搜索空间的一部分。在本节的解决方案中，我们定义了一个包含两种学习算法的搜索空间：逻辑回归和随机森林。每种学习算法都有自己的超参数，并且我们采用 `classifier__[hyperparameter_name]` 这种格式定义超参数的候选值。例如，对于逻辑回归，为了定义正则化超参数空间 `C` 以及可能的正则化惩罚 `penalty` 的候选值集合，创建如下字典：

```
{'classifier': [LogisticRegression()],
 'classifier__penalty': ['l1', 'l2'],
 'classifier__C': np.logspace(0, 4, 10)}
```

对于随机森林的超参数，我们也创建了一个类似的字典：

```
{'classifier': [RandomForestClassifier()],
 'classifier__n_estimators': [10, 100, 1000],
 'classifier__max_features': [1, 2, 3]}
```

在搜索完成之后，可以使用 `best_estimator_` 来查看最佳模型及其超参数：

```
# 查看最佳模型
best_model.best_estimator_.get_params()["classifier"]
```

```
LogisticRegression(C=7.7426368268112693, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=1, penalty='l1', random_state=None,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

和前面两节一样，完成模型选择之后就可以使用这个最佳模型进行预测：

```
# 预测目标向量
best_model.predict(features)
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

12.4 将数据预处理加入模型选择过程

问题描述

将数据预处理步骤纳入模型选择过程。

解决方案

创建一个包含数据预处理步骤及其参数的流水线：

```
# 加载库
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# 设置随机数种子
np.random.seed(0)

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建一个包含 StandardScaler 和 PCA 的预处理对象
preprocess = FeatureUnion([("std", StandardScaler()), ("pca", PCA())])

# 创建一个流水线
pipe = Pipeline([("preprocess", preprocess),
                 ("classifier", LogisticRegression())])

# 创建候选值的取值空间
search_space = [{"preprocess__pca__n_components": [1, 2, 3],
                 "classifier__penalty": ["l1", "l2"],
                 "classifier__C": np.logspace(0, 4, 10)}]

# 创建网格搜索对象
clf = GridSearchCV(pipe, search_space, cv=5, verbose=0, n_jobs=-1)

# 训练模型
best_model = clf.fit(features, target)
```

讨论

使用数据训练模型之前，一般需要对数据进行预处理。在做模型选择时，必须小心地预处理数据。首先，GridSearchCV 使用交叉验证来确定最佳模型。然而在交叉验证中，我们假定用于测试的那一份数据在训练时是不可见的，因此这些数据不应在预处理步骤中（如缩放或标准化）使用。由于上述原因，我们无法先对数据进行预处理，然后再运行 GridSearchCV。相反，预处理步骤必须是 GridSearchCV 操作的一部分。虽然这可能看起来很复杂，但在 scikit-learn 中很容易做到。FeatureUnion 可以组合多个预处理操作。在本节的解决方案中就使用了 FeatureUnion 来组合两个预处理步骤：特征值标准化（StandardScaler）和主成分分析（PCA），生成一个名为 preprocess 的对象，它包含这两个预处理步骤。然后，我们用学习算法将 preprocess 一同包含到流水线中。最终的结果就是，将拟合、转换和使用各种超参数组合训练模型等这些复杂的操作全部交给 scikit-learn 来处理。

其次，一些预处理方法有自己的参数，这些参数通常必须由用户提供。例如，使用 PCA 做降维需要用户定义生成转换特征集的主成分数量。理想情况下，我们会选择能够为指定测试指标生成最佳模型的主成分数量。幸运的是，scikit-learn 也能简化这个操作。当搜索空间中包含预处理参数候选值时，它们会像其他超参数一样被处理。在本解决方案的搜索空间中定义了 features_pca__n_components':[1,2,3],这表明我们想要知道一个、两个或者三个主成分是否可以产生最佳模型。

模型选择结束后，可以查看生成最佳模型的预处理参数。在本解决方案中，可以看到主成分数量：

```
# 查看最佳模型的主成分数量
best_model.best_estimator_.get_params()[ ‘preprocess__pca__n_components’ ]

1
```

12.5 用并行化加速模型选择

问题描述

加快模型选择的过程。

解决方案

设置 n_jobs=-1，以使用所有的 CPU 核：

```
    # 加载库
    import numpy as np
    from sklearn import linear_model, datasets
    from sklearn.model_selection import GridSearchCV

    # 加载数据
    iris = datasets.load_iris()
    features = iris.data
    target = iris.target

    # 创建逻辑回归对象
    logistic = linear_model.LogisticRegression()

    # 创建正则化惩罚的候选超参数区间
    penalty = ["l1", "l2"]

    # 创建参数 C 的候选值区间
    C = np.logspace(0, 4, 1000)

    # 创建超参数选项
    hyperparameters = dict(C=C, penalty=penalty)

    # 创建网格搜索对象
    gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, n_jobs=-1,
verbose=1)

    # 执行网格搜索
    best_model = gridsearch.fit(features, target)

    Fitting 5 folds for each of 2000 candidates, totalling 10000 fits

    [Parallel(n_jobs=-1)]: Done 1352 tasks      | elapsed:    7.2s
    [Parallel(n_jobs=-1)]: Done 5120 tasks      | elapsed:   30.3s
    [Parallel(n_jobs=-1)]: Done 10000 out of 10000 | elapsed:  1.1min finished
```

讨论

在本节的解决方案中，我们将候选模型的数量设置为一个较小值，以便代码可以快速执行完。然而，真实的任务中通常会有数千或数万个模型需要训练，寻找最佳模型的过程可能需要数小时。为了加快这个过程，scikit-learn 允许同时训练多个模型。在不涉及太多技术细节的情况下，scikit-learn 可以同时训练的模型数量等于机器的 CPU 核的数量。大多数现代笔记本电脑都有 4 个核，因此你可以同时训练 4 个模型（假设你目前使用的是笔记本电脑）。这将大大加快模型选择的速度。参数 `n_jobs` 定义了并行训练的模型数量。

解决方案中将 n_jobs 设置为 -1，表示 scikit-learn 将使用所有 CPU 核。然而默认情况下，n_jobs 的值为 1，表示只使用一个核。为了证明这一点，我们运行与解决方案相同的 GridSearchCV，但是 n_jobs=1，你可以看到寻找最佳模型所需的时间明显变长了（注意，确切的时间与你的计算机有关）：

```
# 创建使用单个核的网格搜索
clf = GridSearchCV(logistic, hyperparameters, cv=5, n_jobs=1, verbose=1)

# 执行网格搜索
best_model = clf.fit(features, target)

Fitting 5 folds for each of 2000 candidates, totalling 10000 fits

[Parallel(n_jobs=1)]: Done 10000 out of 10000 | elapsed:  2.8min finished
```

12.6 使用针对特定算法的方法加速模型选择

问题描述

要加快模型选择的过程。

解决方案

如果你使用的是特定的学习算法，可以通过 scikit-learn 的模型特有的交叉验证超参数调优功能来加速模型选择。例如，使用 LogisticRegressionCV：

```
# 加载库
from sklearn import linear_model, datasets

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建 LogisticRegressionCV 对象
logit = linear_model.LogisticRegressionCV(Cs=100)

# 训练模型
logit.fit(features, target)

LogisticRegressionCV(Cs=100, class_weight=None, cv=None, dual=False,
           fit_intercept=True, intercept_scaling=1.0, max_iter=100,
           multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
```

```
        refit=True, scoring=None, solver='lbfgs', tol=0.0001, verbose=0)
```

讨论

有时，利用学习算法的特性能够比暴力搜索或随机模型搜索更快地找到最佳超参数。在scikit-learn中，许多学习算法（如岭回归、套索回归和弹性网络回归）都有特定的交叉验证方法来利用其自身的优势寻找最佳超参数。例如，`LogisticRegression`用于执行标准逻辑回归分类器，而`LogisticRegressionCV`则实现了一个高效的交叉验证逻辑回归分类器，能够识别超参数C的最佳值。

scikit-learn的`LogisticRegressionCV`方法包含一个`Cs`参数。如果`Cs`为一个列表，就可以从`Cs`中选择候选超参数。如果`Cs`为一个整数，就会生成一个含有对应数量候选值的列表。这些候选值按照对数值间隔相等的方式，在0.0001到10,000之间（C的合理取值范围）抽取。

但是，`LogisticRegressionCV`有一个缺点，它只能搜索C的取值区间。在12.1节中，超参数空间包括C和另一个超参数（正则化惩罚）。对于scikit-learn中许多模型特有的交叉验证方法而言，这种限制是很常见的。

延伸阅读

- scikit-learn文档：LogisticRegressionCV（*http://bit.ly/2GPJvjY*）
- scikit-learn文档：模型特有的交叉验证（*http://bit.ly/2F0TQsL*）

12.7 模型选择后的性能评估

问题描述

评估通过模型选择找到的模型的性能。

解决方案

使用嵌套交叉验证来避免评估偏差：

```
# 加载库
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV, cross_val_score
```

```
    # 加载数据
    iris = datasets.load_iris()
    features = iris.data
    target = iris.target

    # 创建逻辑回归对象
    logistic = linear_model.LogisticRegression()

    # 创建超参数 C 的 20 个候选值
    C = np.logspace(0, 4, 20)

    # 创建超参数字典
    hyperparameters = dict(C=C)

    # 创建网格搜索
    gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, n_jobs=-1,
verbose=0)

    # 执行嵌套交叉验证并输出平均得分
    cross_val_score(gridsearch, features, target).mean()

    0.95343137254901966
```

讨论

嵌套交叉验证对于许多人来说都是一个理解起来有难度的概念。请记住，*k* 折交叉验证会在 *k*-1 折数据上训练模型，而使用剩余的数据对该模型进行预测，然后计算模型的预测值与真实值之间的差距，这个过程会重复 *k* 次。

在本章描述的模型选择方法中（即 GridSearchCV 和 RandomizedSearchCV），都使用了交叉验证来评估哪些超参数生成了最佳模型。然而，这里有一个细微的且通常未被充分认识到的问题：由于我们使用这些数据来选择最佳超参数，因此就不能再使用它们评估模型的性能了。有解决方法吗？在交叉验证中包含另一个用于模型搜索的交叉验证即可！在嵌套交叉验证中，“内部”交叉验证用于选择最佳模型，而“外部”交叉验证对模型性能进行无偏估计。在本解决方案中，进行内部交叉验证的是 GridSearchCV 对象，然后使用 cross_val_score 方法将其封装到外部交叉验证中。

如果你感到困惑，可以做一个简单的实验。首先，设置 verbose = 1，这样就可以看到发生了什么：

```
    gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, verbose=1)
```

然后，执行 gridsearch.fit(features, target)，这就是用于寻找最佳模型的内部交叉

验证：

```
best_model = gridsearch.fit(features, target)

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.4s finished
```

从输出结果可以看到，内部交叉验证对 20 个候选模型分别训练了 5 次，共计 100 个模型。接下来，在新的交叉验证中（默认为 3 折）嵌套 gridsearch：

```
scores = cross_val_score(gridsearch, features, target)

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.5s finished

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.7s finished

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.5s finished
```

输出结果显示，内部交叉验证将 20 个模型训练了 5 次以找到最佳模型，并且使用外部 3 折交叉验证评估该模型，总计训练了 300 个模型。

第 13 章

线性回归

13.0 简介

线性回归是机器学习中最简单的有监督学习算法之一。如果你曾在大学里学过入门级统计学课程，课程的最后一个主题很可能就是线性回归。事实上，由于它非常简单，有时甚至不被认为是机器学习的一部分。无论你是否相信，当目标向量是数值（如房价和年龄）时，线性回归及其扩展一直是常见且有效的做预测的方法。本章将介绍各种线性回归方法（以及一些扩展方法），这些方法可以创建性能不错的预测模型。

13.1 拟合一条直线

问题描述

训练一个能表示特征和目标向量之间线性关系的模型。

解决方案

使用线性回归（scikit-learn 中的 `LinearRegression`）：

```
# 加载库
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston

# 加载只有两个特征的数据集
boston = load_boston()
features = boston.data[:,0:2]
target = boston.target
```

```
# 创建线性回归对象
regression = LinearRegression()

# 拟合线性回归模型
model = regression.fit(features, target)
```

讨论

线性回归假设特征与目标向量之间为近似线性的关系。也就是说，特征对目标向量的影响(也称为*系数*、*权重*或*参数*)是恒定的。为了便于解释，这里仅使用两个特征来训练模型，模型的形式如下：

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \epsilon$$

其中，$\hat{y}$是预测目标，x_i是单个特征的数据，$\hat{\beta}_1$和$\hat{\beta}_2$是通过拟合模型得到的相关系数，ϵ是误差。完成模型拟合之后，可以查看每个参数的值。例如，可以使用 `intercept_` 查看$\hat{\beta}_0$的值（也称为*偏差*或*截距*）：

```
# 查看截距
model.intercept_

22.46681692105723
```

对于$\hat{\beta}_1$和$\hat{\beta}_2$，则可以用 `coef_` 来查看：

```
# 显示特征的权重
model.coef_

array([-0.34977589,  0.11642402])
```

在本数据集中，目标值是波士顿住宅价格(20 世纪 70 年代)的中值，单位为千美元。因此，数据集中第一处住宅的价格是：

```
# 目标向量的第一个值乘以 1000
target[0]*1000

24000.0
```

使用 `predict` 方法，我们可以预测该住宅的价值：

```
# 预测第一个样本的目标值，并乘以 1000
model.predict(features)[0]*1000
```

```
24560.23872370844
```

不错，预测值和真实值只相差 560.24 美元！

线性回归的主要优点是可解释性，因为模型的系数代表特征每单位的变化对目标向量的影响。例如，本解决方案的第一个特征是每个居民的犯罪次数。在我们的模型中，这个特征的系数约为 –0.35，如果将这个系数乘以 1000（因为目标向量是以千美元计数的房价），就可以计算人均犯罪每增加一次以后房价的变化：

```
# 第一个系数乘以 1000
model.coef_[0]*1000
```

```
-349.77588707748947
```

这表示人均犯罪量每增加 1 次会使房屋价格降低约 350 美元！

13.2 处理特征之间的影响

问题描述

一个特征对目标变量的影响取决于另一个特征，现在要处理两个特征之间的影响。

解决方案

使用 scikit-learn 的 `PolynomialFeatures` 创建多项式特征，对这种依赖关系建模：

```
# 加载库
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston
from sklearn.preprocessing import PolynomialFeatures

# 加载只有两个特征的数据集
boston = load_boston()
features = boston.data[:,0:2]
target = boston.target

# 创建交互特征
interaction = PolynomialFeatures(
    degree=3, include_bias=False, interaction_only=True)
features_interaction = interaction.fit_transform(features)

# 创建线性回归对象
regression = LinearRegression()
```

```
# 拟合线性回归
model = regression.fit(features_interaction, target)
```

讨论

有时某个特征对目标向量的影响部分取决于另一个特征。想象一个与冲咖啡有关的简单例子。我们有两个二元特征——是否加糖（sugar）和是否搅拌（stirred），现在想要预测咖啡是不是甜的。只把糖加入咖啡中（sugar=1，stirred=0）不会使咖啡变甜（所有的糖都沉在底部），只搅拌咖啡而不加糖（sugar=0，stirred=1）也不会让它变甜。但是，把糖放入咖啡中并搅拌（sugar=1，stirred=1）将使咖啡变甜。加糖和搅拌对咖啡变甜的影响是相互依赖的。在这种情况下，可以说加糖和搅拌之间存在相互作用。

将存在相互作用的一组特征的乘积作为新特征加入模型中，就可以对这种特征（交互特征）进行建模，其公式如下：

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \hat{\beta}_3 x_1 x_2 + \epsilon$$

其中，x_1 和 x_2 分别是 sugar 和 stirred 的值，$x_1 x_2$ 代表两者之间的相互作用。

本解决方案使用的是仅包含两个特征的数据集。以下是第一个样本的两个特征：

```
# 查看第一个样本的特征
features[0]
array([  6.32000000e-03,   1.80000000e+01])
```

要创建一个交互特征，只需将每个样本的这两个特征相乘即可：

```
# 加载库
import numpy as np

# 将每个样本的第一个和第二个特征相乘
interaction_term = np.multiply(features[:, 0], features[:, 1])
```

然后，可以查看第一个样本的交互特征：

```
# 查看第一个样本的交互特征
interaction_term[0]

0.11376
```

虽然我们经常有充分的理由相信两个特征之间存在相互作用，但有时并不是这样。在这种情况下，使用 scikit-learn 的 PolynomialFeatures 为所有的特征组合创建交互项（交

互特征）会很有用，然后就可以使用模型选择策略找出能产生最佳模型的特征和交互项的组合。

要使用 PolynomialFeatures 创建交互特征，必须设置 3 个重要的参数。最重要的一个是 interaction_only，设置 interaction_only = True 会告诉 PolynomialFeatures 只返回交互特征（而不是多项式特征，对于多项式特征，我们将在 13.3 节中讨论）。默认情况下，PolynomialFeatures 将添加一个叫作偏差的特征（值全部为 1），我们可以使用 include_bias = False 来阻止加入偏差。最后，degree 参数用于确定最多用几个特征来创建交互特征（允许创建包含三个特征的交互特征）。在本解决方案中，可以通过观察第一个样本的特征与交互特征是否与我们人工计算的结果相同，来检查 PolynomialFeatures 是否正常工作：

```
# 观察第一个样本的值
features_interaction[0]
```

```
array([  6.32000000e-03,   1.80000000e+01,   1.13760000e-01])
```

13.3 拟合非线性关系

问题描述

对一个非线性关系建模。

解决方案

在线性回归模型中纳入多项式特征，以创建多项式回归模型：

```
# 加载库
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston
from sklearn.preprocessing import PolynomialFeatures

# 加载只有一个特征的数据集
boston = load_boston()
features = boston.data[:,0:1]
target = boston.target

# 创建多项式特征 x^2 和 x^3
polynomial = PolynomialFeatures(degree=3, include_bias=False)
features_polynomial = polynomial.fit_transform(features)

# 创建线性回归对象
```

```
regression = LinearRegression()

# 拟合线性回归模型
model = regression.fit(features_polynomial, target)
```

讨论

到目前为止，我们只讨论了对线性关系的建模。建筑物的楼层数和建筑物的高度就是线性关系的一个例子。在线性回归中，我们假设楼层数量和建筑物高度之间的关系大致是恒定的，也就是说 20 层楼的高度大约是 10 层楼的两倍，10 层楼的高度大约是 5 层楼的两倍。但是有许多关系并不是严格线性的。

我们也会希望拟合一个非线性关系，比如学生学习的小时数与他的考试分数之间的关系。从直觉上，我们可以想象完全不学习的学生与学习 1 小时的学生考试分数存在很大差异。但是，对于学习 99 小时的学生和学习 100 小时的学生来说，他们之间的分数差异要小很多，1 小时的学习时间对学生考试成绩的影响随着学习总时长的增加而减小。

多项式回归是线性回归的一种扩展，它使我们可以对非线性关系进行建模。要创建多项式回归模型，需要将 13.1 节中使用的线性函数：

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \epsilon$$

转换成多项式函数，这可以通过加入多项式特征来实现：

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_1^2 + \ldots + \hat{\beta}_d x_i^d + \epsilon$$

其中 d 是多项式的维度。为什么可以将线性回归用于非线性函数？因为这里只是为模型添加了特征，并没有改变线性回归拟合模型的方式。也就是说，线性回归并不“知道”x^2 是 x 的二次变换，而是把它当作另一个变量来处理的。

下面对这一过程进行分步介绍。首先，为了建模非线性关系，我们可以将现有特征提升到某个幂次（平方项、三次方等）来创建新特征。模型添加的新特征越多，拟合的“线”就越灵活。为了进一步说明，这里假设需要创建一个最高到三阶的多项式。简单起见，我们只关注一个样本（数据集中的第一个样本）x_0：

```
# 观察第一个样本
features[0]

array([ 0.00632])
```

为了创建多项式特征，将第一个样本的值提升到二阶（x_1^2）：

```
# 观察第一个样本的平方值
features[0]**2

array([  3.99424000e-05])
```

这样就得到了一个新特征。然后将第一个样本的值提升到三阶（x_1^3）：

```
# 观察第一个样本的三次方值
features[0]**3

array([  2.52435968e-07])
```

将所有三个特征（x、x^2 和 x^3）包含在特征矩阵中，然后进行线性回归，这样就构造了一个多项式回归模型：

```
# 观察第一个样本的所有三个特征 x、x 的平方和 x 的 3 次方
features_polynomial[0]

array([  6.32000000e-03,   3.99424000e-05,   2.52435968e-07])
```

`PolynomialFeatures` 有两个重要参数。第一个参数是 `degree`，它确定多项式特征的最高阶数。例如，`degree = 3` 将生成 x^2 和 x^3。第二个参数是 `include_bias`，默认情况下 `PolynomialFeatures` 包含一个全为 1 的特征（称为偏差，bias），可以通过设置 `include_bias = False` 来删除它。

13.4 通过正则化减少方差

问题描述

希望减少线性回归模型的方差。

解决方案

使用包含*惩罚项*（也称为*正则化项*）的学习算法，如岭回归（ridge regression）和套索回归（lasso regression）：

```
# 加载库
from sklearn.linear_model import Ridge
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler

# 加载数据
```

```
boston = load_boston()
features = boston.data
target = boston.target

# 特征标准化
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# 创建一个包含指定 alpha 值的岭回归
regression = Ridge(alpha=0.5)

# 拟合线性回归模型
model = regression.fit(features_standardized, target)
```

讨论

在标准线性回归中，通过最小化真实值（y_i）和预测值（$\hat{y}_i$）的平方误差来训练模型，这个平方误差值也被称为残差平方和（RSS，Residual Sum Of Squares）：

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

正则化的回归模型与其相似，但是它在优化对象 RSS 上加入了对系数值的惩罚，该惩罚项被称为收缩惩罚（shrinkage penalty），因为它试图“缩小”模型。线性回归有两种常见的正则化优化方法：岭回归和套索回归。二者的差异在于所使用的收缩惩罚项不同。在岭回归中，收缩惩罚项是可调超参数与所有系数的平方和的乘积：

$$\text{RSS}+\alpha \sum_{j=1}^{p} \hat{\beta}_j^2$$

其中，$\hat{\beta}_j$是总计 p 个特征中第 j 个特征的系数，α 是超参数（下面会讨论）。套索回归与此相似，只不过收缩惩罚项变成了可调超参数与所有系数绝对值之和的乘积：

$$\frac{1}{2n}\text{RSS} + \alpha \sum_{j=1}^{p} \left|\hat{\beta}_j\right|$$

其中，n 是样本的数量。那么在使用时我们应该选择哪一种？根据经验，岭回归通常比套索回归产生的结果稍好，但是套索回归（将在 13.5 节中讨论）产生的模型更容易解释。如果想在岭回归和套索回归的惩罚项之间折中，可以使用弹性网络（elastic net），它是一种同时包含了两种惩罚项的回归模型。无论使用哪一种惩罚项，岭回归和套索回归都

可以通过在损失函数中加入$\hat{\beta}_j$来惩罚大型或复杂模型。

超参数 α 用来控制对$\hat{\beta}_j$的惩罚强度，α 值越大，生成的模型就越简单。α 的理想值应该是像其他超参数一样通过调试获得的。在 scikit-learn 中，使用 alpha 参数来设置 α。

scikit-learn 包含一个 RidgeCV 方法，可以使用它来选择理想的值：

```
# 加载库
from sklearn.linear_model import RidgeCV


# 创建包含三个alpha值的RidgeCV对象
regr_cv = RidgeCV(alphas=[0.1, 1.0, 10.0])

# 拟合线性回归
model_cv = regr_cv.fit(features_standardized, target)

# 查看模型的系数
model_cv.coef_

array([-0.91215884,  1.0658758 ,  0.11942614,  0.68558782, -2.03231631,
        2.67922108,  0.01477326, -3.0777265 ,  2.58814315, -2.00973173,
       -2.05390717,  0.85614763, -3.73565106])
```

还可以查看最优模型的值：

```
# 查看 alpha 值
model_cv.alpha_


1.0
```

最后要注意一点：在线性回归中系数的值受特征的范围（scale）的影响，而在正则化模型中所有系数会被加在一起，所以在训练模型之前必须确保特征已经标准化。

13.5 使用套索回归减少特征

问题描述

希望通过减少特征的数量来简化线性回归模型。

解决方案

使用套索回归：

```
# 加载库
from sklearn.linear_model import Lasso
from sklearn.datasets import load_boston
```

```
from sklearn.preprocessing import StandardScaler

# 加载数据
boston = load_boston()
features = boston.data
target = boston.target

# 特征标准化
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# 创建套索回归，并指定 alpha 值
regression = Lasso(alpha=0.5)

# 拟合线性回归
model = regression.fit(features_standardized, target)
```

讨论

套索回归的惩罚项可以将特征的系数减小为零，从而有效地减少模型中特征的数量。在本解决方案中 alpha 值被设置为 0.5，可以看到许多系数为 0，这意味着它们对应的特征并未在模型中使用：

```
# 查看系数
model.coef_

array([-0.10697735,  0.        , -0.        ,  0.39739898, -0.        ,
        2.97332316, -0.        , -0.16937793, -0.        , -0.        ,
       -1.59957374,  0.54571511, -3.66888402])
```

如果将 alpha 设为一个更大的值，会看到模型不会使用任何特征：

```
# 创建一个alpha值为10的套索回归
regression_a10 = Lasso(alpha=10)
model_a10 = regression_a10.fit(features_standardized, target)

model_a10.coef_
array([-0.,  0., -0.,  0., -0.,  0., -0.,  0., -0., -0., -0.,  0., -0.])
```

利用这种特性，我们可以在特征矩阵中包含 100 个特征，然后调整套索回归的超参数，生成比如仅使用 10 个最重要特征的模型。这样做可以减少模型方差，同时提高模型的可解释性（特征越少就越容易解释）。

第 14 章

树和森林

14.0 简介

基于树的学习算法是十分流行且应用广泛的一类非参数化的有监督学习算法，这些算法既可用于分类又可用于回归。基于树的学习算法的基础是包含一系列决策规则（例如，“如果他们是男性……”）的决策树。这些决策规则看起来很像一棵倒置的树，第一个决策规则在顶部，随后的决策规则在其下面展开。在决策树中，每个决策规则产生一个决策节点，并创建通向新节点的分支。终点处没有决策规则的分支被称为叶子节点（*leaf*）。

决策树型模型普及的一个原因是它们的可解释性很强。实际上，通过绘制完整的决策树可以创建一个非常直观的模型（参见 14.3 节）。从这个基本的树系统可以引出各种各样的扩展，包括随机森林和堆叠（stacking）模型。本章将介绍如何训练、处理、调整、可视化和评估基于树的模型。

14.1 训练决策树分类器

问题描述

使用决策树训练分类器。

解决方案

使用 scikit-learn 中的 `DecisionTreeClassifier`：

```
# 加载库
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets
```

```
# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建决策树分类器对象
decisiontree = DecisionTreeClassifier(random_state=0)

# 训练模型
model = decisiontree.fit(features, target)
```

讨论

决策树的训练器会尝试找到在节点上能够最大限度降低数据不纯度（impurity）的决策规则。度量不纯度的方式有许多，`DecisionTreeClassifier`默认使用基尼不纯度（Gini impurity）：

$$G(t) = 1 - \sum_{i=1}^{c} p_i^2$$

其中，$G(t)$是节点 t 的基尼不纯度，p_i 是在节点 t 上第 i 类样本的比例。寻找使不纯度降低的决策规则的过程会被递归执行，直到所有叶子节点都变为纯节点（即仅包含一种分类）或达到某个终止条件。

在 scikit-learn 中，`DecisionTreeClassifier`的使用方式与其他学习算法类似，首先用 `fit`方法训练模型，然后就可以用训练好的模型来预测一个样本的分类：

```
# 创建新样本
observation = [[ 5, 4, 3, 2]]

# 预测样本的分类
model.predict(observation)

array([1])
```

也可以使用 `predict_proba`方法查看该样本属于每个分类（预测的分类）的概率：

```
# 查看样本分别属于三个分类的概率
model.predict_proba(observation)

array([[ 0., 1., 0.]])
```

最后，如果想使用其他的不纯度度量方式，可以修改参数 `criterion`：

```
# 使用 entropy 作为不纯度检测方法创建决策树分类器对象
decisiontree_entropy = DecisionTreeClassifier(
    criterion='entropy', random_state=0)

# 训练模型
model_entropy = decisiontree_entropy.fit(features, target)
```

延伸阅读

- 《决策树学习》（Princeton 大学官网，*http://bit.ly/2FqJxlj*）

14.2 训练决策树回归模型

问题描述

训练一个基于决策树的回归模型。

解决方案

使用 scikit-learn 中的 `DecisionTreeRegressor`：

```
# 加载库
from sklearn.tree import DecisionTreeRegressor
from sklearn import datasets

# 加载仅有两个特征的数据
boston = datasets.load_boston()
features = boston.data[:,0:2]
target = boston.target

# 创建决策树回归模型对象
decisiontree = DecisionTreeRegressor(random_state=0)

# 训练模型
model = decisiontree.fit(features, target)
```

讨论

决策树回归模型与决策树分类模型的工作方式类似，不过前者不会使用基尼不纯度或熵的概念，而是默认使用均方误差（MSE）的减少量来作为分裂规则的评估标准：

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2$$

其中，y_i是样本的真实值，$\hat{y}_i$是样本的预测值。在 scikit-learn 中，决策树回归模型可以用 DecisionTreeRegressor 构建。模型训练完以后可以用它对样本的值进行预测：

```
# 创建新样本
observation = [[0.02, 16]]

# 预测样本值
model.predict(observation)

array([ 33.])
```

同决策树分类器 DecisionTreeClassifier 一样，用参数 criterion 可以选择分裂质量（split quality）的度量方式。例如，可以用平均绝对误差（MAE）的减少量作为分裂标准来构造决策树模型：

```
# 使用 MAE 创建决策树回归模型
decisiontree_mae = DecisionTreeRegressor(criterion="mae", random_state=0)

# 训练模型
model_mae = decisiontree_mae.fit(features, target)
```

延伸阅读

- scikit-learn 文档：决策树回归（*http://bit.ly/2GR63AZ*）

14.3 可视化决策树模型

问题描述

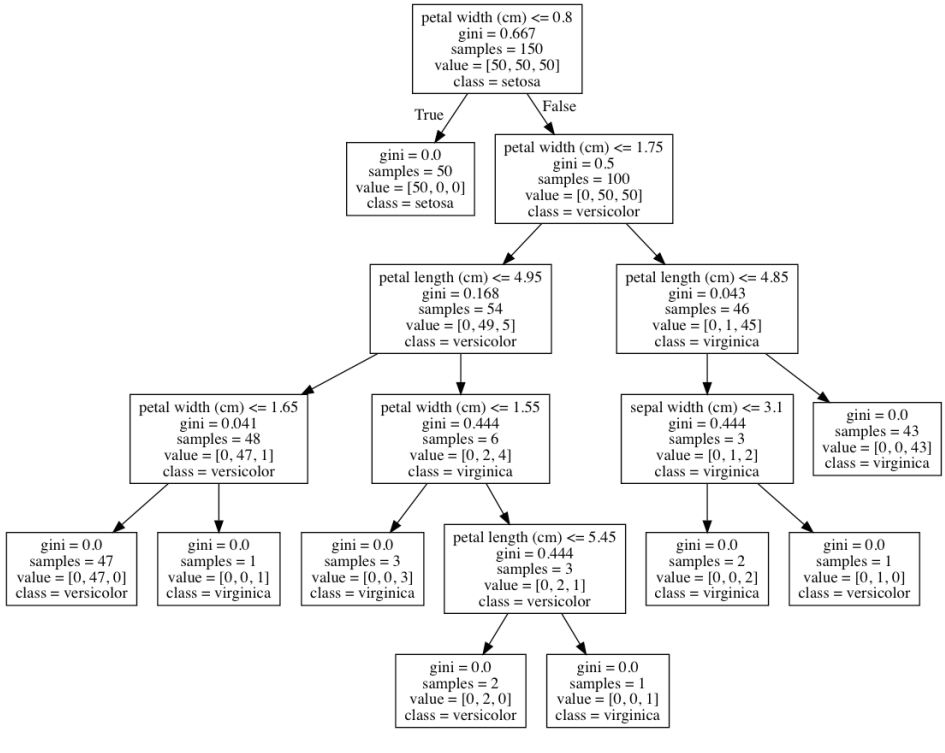
可视化一个决策树模型。

解决方案

将决策树模型导出为 DOT 格式并可视化：

```
# 加载库
import pydotplus
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets
from IPython.display import Image
from sklearn import tree
```

```
# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建决策树分类器对象
decisiontree = DecisionTreeClassifier(random_state=0)

# 训练模型
model = decisiontree.fit(features, target)

# 创建 DOT 数据
dot_data = tree.export_graphviz(decisiontree,
                                out_file=None,
                                feature_names=iris.feature_names,
                                class_names=iris.target_names)

# 绘制图形
graph = pydotplus.graph_from_dot_data(dot_data)

# 显示图形
Image(graph.create_png())
```



讨论

可以将整个模型可视化，是决策树分类器的优点之一，这也使决策树成为机器学习中解释性最好的模型之一。在本解决方案中，模型以 DOT 格式（一种图形描述语言）导出，然后被绘制成图形。

如果查看根节点，可以看到决策规则是，如果花瓣宽度小于或等于 0.8 就转到左分支，否则就转到右分支。同时，我们还可以看到基尼不纯度指数为 `0.667`，样本数量为 `150`，每个分类中的样本数量分布为 `[50,50,50]`，如果在该节点停止，将得到样本的预测分类为 `setosa`。我们还可以知道在该节点处，依靠单个决策规则（`petal width (cm) <= 0.8`）能够完美地识别所有的 `setosa` 类样本。此外，添加一个基于同一个特征的决策规则（`petal width (cm) <= 1.75`），决策树能够对 150 个样本中的 144 个正确分类。这使得花瓣宽度成为一个非常重要的特征！

如果要在其他应用或者报告中使用该决策树，可以将可视化后的决策树导出为 PDF 格式或 PNG 格式：

```
# 创建 PDF
graph.write_pdf("iris.pdf")

True

# 创建 PNG
graph.write_png("iris.png")

True
```

本节的解决方案对决策树分类模型进行了可视化操作，这种方法也可以用于决策树回归模型。

注意：macOS 用户可能需要安装 GraphViz 才能运行上述代码。可以使用 Homebrew 命令（`brew install graphviz`）进行安装。有关 Homebrew 的安装说明，请访问 Homebrew 的官方网站。

延伸阅读

- Homebrew（*https://brew.sh*）

14.4 训练随机森林分类器

问题描述

训练一个随机森林分类器模型。

解决方案

使用 scikit-learn 中的 RandomForestClassifier 训练随机森林分类器模型：

```
# 加载库
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建随机森林分类器对象
randomforest = RandomForestClassifier(random_state=0, n_jobs=-1)

# 训练模型
model = randomforest.fit(features, target)
```

讨论

决策树有一个常见问题，即倾向于紧密地拟合训练数据（过拟合）。这使得随机森林这种集成学习方法被普遍应用。在随机森林中，许多决策树同时被训练，但是每棵树只接收一个自举的（bootstrapped）样本（即有放回的随机抽样，抽样次数与原始样本数相同），并且每个节点在确定最佳分裂时只考虑全部特征的一个子集。这个由随机决策树组成的森林（随机森林因此而得名）通过投票决定样本的预测分类。

将此解决方案与 14.1 节的进行比较，可以发现 scikit-learn 的 RandomForestClassifier 与 DecisionTreeClassifier 的工作方式类似：

```
# 创建新样本
observation = [[ 5, 4, 3, 2]]

# 预测样本的分类
model.predict(observation)

array([1])
```

RandomForestClassifier 也使用与 DecisionTreeClassifier 相同的参数。例如，我们可以更改度量分裂质量的方法：

```
# 使用熵创建随机森林分类器对象
randomforest_entropy = RandomForestClassifier(
    criterion="entropy", random_state=0)

# 训练模型
model_entropy = randomforest_entropy.fit(features, target)
```

不过，作为一个森林而不是一棵单独的决策树，RandomForestClassifier 有一些独特且重要的参数。首先，参数 max_features 决定每个节点需要考虑的特征的最大数量，允许输入的变量类型包括整型（特征的数量）、浮点型（特征的百分比）和 sqrt（特征数量的平方根）。默认情况下，参数 max_features 的值被设置为 auto（相当于 sqrt）。其次，参数 bootstrap 用于设置在创建树时使用的样本子集，是有放回的抽样（默认值）还是无放回的抽样。第三，参数 n_estimators 设置森林中包含的决策树的数量。在 10.4 节中，我们将 n_estimators 作为超参数，并可视化树的数量增加对评估指标的影响。最后，由于需要训练的决策树模型数量众多，可以设置参数 n_jobs = -1 来指定使用所有可用的 CPU 核进行训练，并且该参数不仅仅对随机森林分类器有效。

延伸阅读

- 《随机森林》（Berkeley Statistics，*http://bit.ly/2Fxm0Ps*）

14.5 训练随机森林回归模型

问题描述

要用随机森林训练一个回归模型。

解决方案

使用 scikit-learn 中的 RandomForestRegressor 训练随机森林回归模型：

```
# 加载库
from sklearn.ensemble import RandomForestRegressor
from sklearn import datasets

# 加载仅有两个特征的数据
boston = datasets.load_boston()
features = boston.data[:,0:2]
```

```
target = boston.target

# 创建随机森林回归对象
randomforest = RandomForestRegressor(random_state=0, n_jobs=-1)

# 训练模型
model = randomforest.fit(features, target)
```

讨论

和创建随机森林分类器一样，我们也可以创建随机森林回归模型，其中每棵树使用一个自举的样本子集，并且在每个节点决策规则仅考虑一部分特征。与 RandomForestClassifier 一样，随机森林回归模型也有几个重要的参数：

- max_features 设置每个节点要考虑的特征的最大数量，默认值为$\sqrt{p}$个，其中 p 是特征的总数。
- bootstrap 设置是否使用有放回的抽样，默认值为 True。
- n_estimators 设置决策树的数量，默认值为 10。

延伸阅读

- scikit-learn 文档：RandomForestRegressor（*http://bit.ly/2GQZ3nx*）

14.6 识别随机森林中的重要特征

问题描述

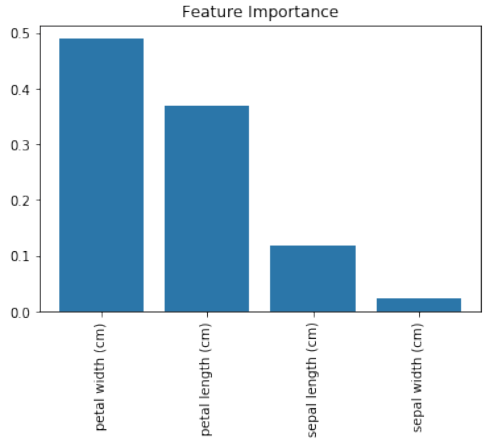
想知道随机森林模型中最重要的特征。

解决方案

计算并可视化每个特征的重要性（Feature importance）：

```
# 加载库
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets
```

```
# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建随机森林分类器对象
randomforest = RandomForestClassifier(random_state=0, n_jobs=-1)

# 训练模型
model = randomforest.fit(features, target)

# 计算特征的重要性
importances = model.feature_importances_

# 将特征的重要性按降序排列
indices = np.argsort(importances)[::-1]

# 按照特征的重要性对特征名称重新排序
names = [iris.feature_names[i] for i in indices]

# 创建图
plt.figure()

# 创建图标题
plt.title("Feature Importance")

# 添加数据条
plt.bar(range(features.shape[1]), importances[indices])

# 将特征名称添加为 x 轴标签
plt.xticks(range(features.shape[1]), names, rotation=90)

# 显示图
plt.show()
```



讨论

决策树的可解释性是它的优点之一，对决策树模型可视化是很容易的（参见 13.3 节）。然而，一个随机森林模型由数十、数百甚至数千棵决策树组成，很难对随机森林模型生成简易直观的可视化结果。不过，我们还可以用另一种方式可视化随机森林：比较（和可视化）每个特征的相对重要性。

在 13.3 节中，我们对决策树分类器模型进行了可视化，并且看到仅基于花瓣宽度的决策规则就能够将大部分样本正确分类，这意味着花瓣宽度是分类器中的一个重要特征。更准确地说，不纯度（比如分类器中的基尼不纯度或熵，以及回归模型中的方差）的平均减少量较大的分裂特征是更重要的特征。

关于特征的重要性有两点需要注意。首先，scikit-learn 需要将 nominal 型分类特征分解为多个二元特征，使得特征的重要性分散到各个二元特征中。这样的话，即使原来的分类特征非常重要，分解后的特征往往也就没那么重要了。其次，如果两个特征高度相关，并且其中一个有很高的重要性，就会使另一个特征的重要性显得稍低，如果不考虑这种情况，模型的效果会受到影响。

在 scikit-learn 中，决策树和随机森林的分类及回归模型都可以通过 `feature_importances_` 查看模型中每个特征的重要程度：

```
# 查看特征的重要程度
model.feature_importances_

array([ 0.11896532, 0.0231668 , 0.36804744, 0.48982043])
```

数值越大，说明该特征越重要（所有特征的重要性分数相加等于 1）。绘制这些值的图形

有助于解释随机森林模型。

14.7 选择随机森林中的重要特征

问题描述

在随机森林中进行特征选择。

解决方案

确定重要的特征，并使用它们重新训练模型：

```
# 加载库
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets
from sklearn.feature_selection import SelectFromModel

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建随机森林分类器
randomforest = RandomForestClassifier(random_state=0, n_jobs=-1)

# 创建对象，选择重要性大于或等于阈值的特征
selector = SelectFromModel(randomforest, threshold=0.3)

# 使用选择器创建新的特征矩阵
features_important = selector.fit_transform(features, target)

# 使用重要的特征训练随机森林模型
model = randomforest.fit(features_important, target)
```

讨论

在某些情况下，你可能希望减少模型中特征的数量。例如，想减少模型的方差或者希望仅使用少数重要的特征来提高模型的可解释性。

在 scikit-learn 中，可以使用一个简单的两步工作流来创建一个使用较少特征的模型。首先，使用所有特征训练一个随机森林模型，并使用训练得到的模型来确定重要的特征。接下来，创建一个仅包含这些重要特征的新特征矩阵。本节的解决方案使用了

SelectFromModel 方法来创建特征矩阵，其中仅包含重要性大于或等于某阈值的特征。最后，使用这些特征创建一个新模型。

这种方法中有两点需要注意：第一，经过 one-hot 编码的 nominal 型分类特征的重要性被稀释到二元特征中；第二，一对高度相关的特征，其重要性被集中在其中一个特征上，而不是均匀分布在这两个特征上。

延伸阅读

- 《使用随机森林进行特征选择》（Robin Genuer、Jean-Michel Poggi、Christine Tuleau-Malot，*http://bit.ly/2FvG70D*）

14.8 处理不均衡的分类

问题描述

在高度不均衡的数据上训练随机森林模型。

解决方案

用参数 class_weight="balanced" 训练决策树或随机森林模型：

```
# 加载库
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 删除前 40 个样本以获得高度不均衡的数据
features = features[40:,:]
target = target[40:]

# 创建目标向量表明分类为 0 还是 1
target = np.where((target == 0), 0, 1)

# 创建随机森林分类器对象
randomforest = RandomForestClassifier(
    random_state=0, n_jobs=-1, class_weight="balanced")
```

```
# 训练模型
model = randomforest.fit(features, target)
```

讨论

在现实中用学习算法训练模型时，很容易遇到不均衡的分类问题。如果不解决这个问题，不均衡的分类就会降低模型的性能。我们在 17.5 节将讨论在数据预处理过程中处理不均衡分类的一些策略。不过在 scikit-learn 中，很多学习算法都带有用于纠正不均衡分类的内置方法。为 `RandomForestClassifier` 设置参数 `class_weight` 就可以纠正不均衡分类的问题。如果将分类名和所需的权重以字典形式提供（如 `{"male": 0.2,"female": 0.8}`），`RandomForestClassifier` 将为各个分类相应地加权。不过参数 `balanced` 通常更有用，它根据各个分类在数据中出现频率的倒数自动计算权重值：

$$w_j = \frac{n}{kn_j}$$

其中，w_j 是分类 j 的权重，n 是样本数量，n_j 是分类 j 中样本的数量，k 是分类的总数。举例来说，前面的解决方案中有两种分类（k），110 个样本（n），每个分类中的样本数分别为 10 和 100（n_j）。如果使用参数 `class_weight="balanced"`，就能够增大较小分类的权重（减小较大分类的权重）：

```
# 计算较小分类的权重
110/(2*10)
```

```
5.5
```

较大分类的权重值比较小：

```
# 计算较大分类的权重
110/(2*100)
```

```
0.55
```

14.9 控制决策树的规模

问题描述

手动控制决策树的结构和规模。

解决方案

使用 scikit-learn 控制基于树的算法中的结构参数：

```
# 加载库
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建决策树分类器对象
decisiontree = DecisionTreeClassifier(random_state=0,
                                      max_depth=None,
                                      min_samples_split=2,
                                      min_samples_leaf=1,
                                      min_weight_fraction_leaf=0,
                                      max_leaf_nodes=None,
                                      min_impurity_decrease=0)

# 训练模型
model = decisiontree.fit(features, target)
```

讨论

在 scikit-learn 基于树的学习算法中，有很多用于控制决策树规模的方法。我们可以通过设置不同的参数来访问这些方法。

`max_depth`：

> 树的最大深度。如果该参数为 `None`，则树将一直生长，直到所有叶子都为纯节点。如果提供整数作为该参数的值，这棵树就会被有效“修剪”到这个整数值所表示的深度。

`min_samples_split`

> 在该节点分裂之前，节点上最小的样本数。如果提供整数作为该参数的值，则这个整数代表最小的样本数；如果提供浮点数作为参数值，则最小样本数为总样本数乘以该浮点数。

`min_samples_leaf`

> 叶子节点需要的最小样本数。与 `min_samples_split` 使用相同的参数格式。

`max_leaf_nodes`

最大叶子节点数。

`min_impurity_split`

执行分裂所需的最小不纯度减少量。

了解这些参数很有用，不过一般情况下，我们只会用到 `max_depth` 和 `min_impurity_split` 这两个参数，因为较浅的树（有时称为*树桩*，*stump*）结构更简单，并且方差较小。

14.10 通过 boosting 提高性能

问题描述

需要一个比决策树或随机森林性能更好的模型。

解决方案

使用 `AdaBoostClassifier` 或 `AdaBoostRegressor` 训练一个 boosting 模型：

```
# 加载库
from sklearn.ensemble import AdaBoostClassifier
from sklearn import datasets

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建 adaboost 树分类器对象
adaboost = AdaBoostClassifier(random_state=0)

# 训练模型
model = adaboost.fit(features, target)
```

讨论

在随机森林中，用一组随机决策树对目标向量进行预测。而一种替代的且更强大的学习方法称为 *boosting*。有一种形式的 boosting 算法叫作 AdaBoost，它迭代地训练一系列弱模型（通常是一个浅层决策树，有时称为树桩），每次迭代都会为前一个模型预测错的样本分配更大的权重。AdaBoost 算法的具体步骤如下：

1．为每个样本 x_i 分配初始权重值 $w_i = \frac{1}{n}$，其中 n 是数据中样本的总数。

2．用数据训练弱模型。

3．对于每一个样本：

（1）如果弱模型对 x_i 的预测是正确的，则 w_i 减少。

（2）如果弱模型对 x_i 的预测是错误的，则 w_i 增大。

4．训练一个新的弱模型，样本拥有更大的 w_i，优先级更高。

5．重复步骤 3 和 4，直到数据被完美预测或已经训练完预定数量的弱模型。

最终结果是一个组合模型，里面的不同弱模型聚焦于（从预测角度来看）更复杂的样本。在 scikit-learn 中，我们可以使用 `AdaBoostClassifier` 或 `AdaBoostRegressor` 实现 AdaBoost。其最重要的参数是 `base_estimator`、`n_estimators` 和 `learning_rate`。

- `base_estimator` 表示训练弱模型的学习算法。这个参数几乎不需要改变，因为到目前为止决策树（默认值）是 AdaBoost 最常用的学习算法。
- `n_estimators` 是需要迭代训练的模型数量。
- `learning_rate` 是每个弱模型的权重变化率，默认值为 1。减小这个参数值意味着权重变化的幅度变小，这会使模型的训练速度变慢（但有时会使模型的性能更好）。
- `loss` 是 `AdaBoostRegressor` 独有的参数，它设置了在更新权重时所用的损失函数。其默认值为线性损失函数，但是可以改为平方（`square`）或指数函数（`exponential`）。

延伸阅读

- 《AdaBoost 介绍》（Robert E. Schapire，*http://bit.ly/2FCS30E*）

14.11 使用袋外误差（Out-of-Bag Error）评估随机森林模型

问题描述

在不使用交叉验证的情况下评估随机森林模型。

解决方案

计算模型的袋外误差分数：

```
# 加载库
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建随机数分类器对象
randomforest = RandomForestClassifier(
    random_state=0, n_estimators=1000, oob_score=True, n_jobs=-1)

# 训练模型
model = randomforest.fit(features, target)

# 查看袋外误差
randomforest.oob_score_

0.95333333333333337
```

讨论

在随机森林中，每个决策树使用自举的样本子集进行训练。这意味着对于每棵树而言，都有未参与训练的样本子集。这些样本被称为袋外（Out-of-Bag，OOB）样本。袋外样本可以用作测试集来评估随机森林的性能。

对于每个样本，算法将其真实值与未使用该样本进行训练的树模型子集产生的预测值进行比较。计算所有样本的总得分，就能得到一个随机森林的性能指标。OOB 分数评估法可以作为交叉验证的替代方案。

在 `RandomForestClassifier` 中设置参数 `oob_score = True` 可以计算 OOB 分数。该分数可以使用 `oob_score_` 来获取。

第 15 章

KNN

15.0 简介

KNN（K-Nearest Neighbors，K 近邻）分类器是有监督学习领域中最简单且被普遍使用的分类器之一。KNN 分类器一般被认为是一种懒惰（lazy）的学习器，因为严格来说它并没有训练一个模型用来做预测，而是将观察值的分类判定为离它最近的 k 个观察值中所占比例最大的那个分类。举个例子，如果一个分类未知的观察值周围都是分类为 1 的观察值，那么它会被认为属于分类 1。本章将探索如何使用 scikit-learn 来创建和使用 KNN 分类器。

15.1 找到一个观察值的最近邻

问题描述

找到离一个观察值最近的 k 个观察值（邻居）。

解决方案

使用 scikit-learn 的 `NearestNeighbors`：

```
# 加载库
from sklearn import datasets
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import StandardScaler

# 加载数据
iris = datasets.load_iris()
```

```
features = iris.data

# 创建 standardizer
standardizer = StandardScaler()

# 特征标准化
features_standardized = standardizer.fit_transform(features)

# 两个最近的观察值
nearest_neighbors = NearestNeighbors(n_neighbors=2).fit(features_standardized)

# 创建一个观察值
new_observation = [ 1,  1,  1,  1]

# 获取离观察值最近的两个观察值的索引，以及到这两个点的距离
distances, indices = nearest_neighbors.kneighbors([new_observation])

# 查看最近的两个观察值
features_standardized[indices]

array([[[ 1.03800476,  0.56925129,  1.10395287,  1.1850097 ],
        [ 0.79566902,  0.33784833,  0.76275864,  1.05353673]]])
```

讨论

上面的解决方案中用到了鸢尾花数据集。我们创建了一个观察值 `new_observation`，然后找到了离它最近的两个观察值。`indices` 包含了数据集中最邻近的这两个观察值的位置，所以 `X[indices]` 可以显示它们的数值。从直观上来说，可以把距离看作一种相似度的度量指标，因此这两个最近的观察值代表与我们创建的那朵鸢尾花最相似的两朵鸢尾花。

如何算距离呢？ scikit-learn 提供了很多距离指标（d），包括欧氏距离：

$$d_{euclidean} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

还有曼哈顿距离：

$$d_{manhattan} = \sum_{i=1}^{n}|x_i - y_i|$$

默认情况下，`NearestNeighbors` 使用闵可夫斯基距离：

$$d_{minkowski} = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

这里 x_i 和 y_i 是我们要计算其距离的两个观察值。闵可夫斯基距离中包括一个超参数 p，如果 $p = 1$，其实就是曼哈顿距离；如果 $p = 2$，就是欧氏距离。在 scikit-learn 中，默认 $p = 2$。

我们可以通过 `metric` 参数来设定距离指标：

```
# 找到按照欧氏距离来算最近的两个邻居
nearestneighbors_euclidean = NearestNeighbors(
n_neighbors=2, metric='euclidean').fit(features_standardized)
```

创建的 `distances` 变量中包含到这两个最近邻居的实际距离：

```
# 查看 distances
distances
```

```
array([[ 0.48168828, 0.73440155]])
```

我们还可以用 `kneighbors_graph` 创建一个矩阵，来表示离每个观察值最近的邻居：

```
# 寻找每个观察值按照欧氏距离计算的最近的 3 个邻居（包括它自己）
nearestneighbors_euclidean = NearestNeighbors(
n_neighbors=3, metric="euclidean").fit(features_standardized)

# 每个观察值和它最近的 3 个邻居的列表（包括它自己）
nearest_neighbors_with_self = nearestneighbors_euclidean.kneighbors_graph(
    features_standardized).toarray()

# 从最近邻居的列表里移除自己
for i, x in enumerate(nearest_neighbors_with_self):
    x[i] = 0

# 查看离第一个观察值最近的两个邻居
nearest_neighbors_with_self[0]
```

```
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
```

```
       0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
       0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
       0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

在寻找最近的邻居或者使用基于距离的某种学习算法时，很重要的一件事是转换特征，使所有特征采用同样的单位。这样做是因为距离指标认为所有特征的单位都是相同的，如果一个特征的单位是百万美元，另一个特征的单位是百分比，那么算出来的结果肯定会偏向于前者。我们在解决方案中通过 StandardScaler 来标准化这些特征，解决这种潜在的问题。

15.2 创建一个 KNN 分类器

问题描述

对于分类未知的观察值，基于邻居的分类来预测它的分类。

解决方案

如果数据集不是特别大，就直接用 KNeighborsClassifier：

```
# 加载库
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn import datasets

# 加载数据
iris = datasets.load_iris()
X = iris.data
y = iris.target

# 创建 standardizer
standardizer = StandardScaler()

# 标准化特征
X_std = standardizer.fit_transform(X)

# 训练一个有 5 个邻居的 KNN 分类器
knn = KNeighborsClassifier(n_neighbors=5, n_jobs=-1).fit(X_std, y)

# 创建两个观察值
new_observations = [[ 0.75,  0.75,  0.75,  0.75],
                    [ 1, 1, 1, 1]]
```

```
# 预测这两个观察值的分类
knn.predict(new_observations)

array([1, 2])
```

讨论

在KNN算法中，如果给定一个分类未知的观察值，第一步会先基于某个距离指标（比如欧氏距离）找到最近的 k 个观察值（有时候称为 x_u 的邻域），然后这 k 个观察值基于它们自己的分类来“投票”，得票最多的那个分类就是预测的分类。更正式一点的表述是，属于某个分类 j 的概率是：

$$\frac{1}{k}\sum_{i \in v} I(y_i = j)$$

这里 v 是 x_u 的邻域内的 k 个观察值，y_i 是第 i 个观察值的分类，I 是一个指示函数（即函数值为1就是真，为0就是假）。在scikit-learn中，我们可以使用 `predict_proba` 来查看这些概率：

```
# 查看每个观察值分别属于3个分类中的某一个的概率
knn.predict_proba(new_observations)
array([[ 0. ,  0.6,  0.4],
       [ 0. ,  0. ,  1. ]])
```

概率最高的分类就是我们的预测分类。举个例子，在上面的输出结果中，第一个观察值应该属于分类1（$Pr = 0.6$），而第二个观察值应该属于分类2（$Pr = 1$），如下所示：

```
knn.predict(new_observations)
array([1, 2])
```

`KNeighborsClassifier` 中有一些参数值得注意。第一个是 `metric`，它可以用来设定使用何种距离指标（详见14.1节）。第二个是 `n_jobs`，它控制可以使用多少个CPU内核。做预测需要计算一个点到数据集中所有点的距离，所以笔者强烈建议使用多个内核。第三个是 `algorithm`，它用来设定计算最近邻居的算法。尽管不同的算法之间有很大的区别，但是 `KNeighborsClassifier` 默认会自动选择最合适的算法，所以一般不太需要为怎么选择这个参数而操心。默认情况下 `KNeighborsClassifier` 就是像我们前面描述的那样工作的，邻域内的每个观察值都可以投一票，用来确定预测分类，但是如果我们给 `distance` 设定了 `weights` 参数，那么距离近的观察值的投票比距离远的观察值的投票会有更高的权重。这种设定从直观上来说是有道理的，因为距离更近的邻居可能会告诉我们更多关于观察值分类的信息。

最后，正如 15.1 节中讨论的，因为计算距离时所有的特征被认为是在同一单位下的，所以在使用 KNN 分类器之前标准化特征是很重要的。

15.3 确定最佳的邻域点集的大小

问题描述

为 KNN 分类器找到最佳的 *k* 值。

解决方案

使用 GridSearchCV 这样的模型选择技术：

```
# 加载库
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.model_selection import GridSearchCV

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建 standardizer
standardizer = StandardScaler()

# 标准化特征
features_standardized = standardizer.fit_transform(features)

# 创建一个 KNN 分类器
knn = KNeighborsClassifier(n_neighbors=5, n_jobs=-1)

# 创建一个流水线
pipe = Pipeline([("standardizer", standardizer), ("knn", knn)])

# 确定一个可选值的范围
search_space = [{"knn__n_neighbors": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}]

# 创建 grid 搜索
classifier = GridSearchCV(
    pipe, search_space, cv=5, verbose=0).fit(features_standardized, target)
```

讨论

k 值的大小对 KNN 分类器的性能是有重要影响的。在机器学习中，我们一直尝试在偏差（bias）和方差（variance）之间找到一种平衡，而 k 值对这种平衡的影响很明显。如果 $k = n$（这里 n 是观察值的数量），那么偏差就会很大而方差很小。如果 $k = 1$，那么偏差会很小，但是方差很大。只有找到了能在偏差和方差之间取得折中的 k 值，才能得到最佳的 KNN 分类器。在解决方案中，我们用 `GridSearchCV` 对不同 k 值的 KNN 分类器做 5 折交叉验证。当这个过程结束时，就可以得到能产生最佳 KNN 分类器的 k 值：

```
# 最佳邻域的大小（k）
classifier.best_estimator_.get_params()["knn__n_neighbors"]
```

```
6
```

15.4 创建一个基于半径的最近邻分类器

问题描述

对于分类未知的观察值，根据一定距离范围内所有观察值的分类来确定其分类。

解决方案

使用 `RadiusNeighborsClassifier`：

```
# 加载库
from sklearn.neighbors import RadiusNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn import datasets

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建 standardizer
standardizer = StandardScaler()

# 标准化特征
features_standardized = standardizer.fit_transform(features)

# 训练一个基于半径的最近邻分类器
rnn = RadiusNeighborsClassifier(
    radius=.5, n_jobs=-1).fit(features_standardized, target)
```

```
# 创建两个观察值
new_observations = [[ 1,  1,  1,  1]]

# 预测这两个观察值的分类
rnn.predict(new_observations)

array([2])
```

讨论

在 KNN 分类器中，观察值的分类是根据它的 k 个邻居的分类来预测的；而在不太常用的基于半径的最近邻分类器中，观察值的分类是根据某一半径 r 范围内所有观察值的分类来预测的。在 scikit-learn 中，`RadiusNeighborsClassifier` 与 `KNeighborsClassifier` 很相似，除了两个参数。第一个参数是 `radius`，在 `RadiusNeighborsClassifier` 中，我们需要指定一个半径来确定某个观察值能不能算作目标观察值的邻居。除非你有很充分的理由要把 `radius` 设为某个值，否则最好像对待其他的超参数一样在模型选择期间对它进行调整。第二个有用的参数是 `outlier_label`，它用来指定如果一个观察值周围没有其他观察值在半径 `radius` 的范围内，这个观察值应该被标记为什么。这是一个有用的分辨界外点的方法。

第 16 章

逻辑回归

16.0 简介

尽管名字里有“回归”两个字，但逻辑回归其实是一个被广泛使用的有监督分类方法。逻辑回归和它的一些扩展（比如多元逻辑回归）让我们可以通过一个简单易懂的方法来预测一个观察值属于某个分类的概率。本章将讲解如何使用 scikit-learn 来训练各种分类器。

16.1 训练二元分类器

问题描述

训练一个简单的二元分类器模型。

解决方案

使用 scikit-learn 的 `LogisticRegression` 训练一个逻辑回归模型：

```
 # 加载库
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# 加载仅有两个分类的数据
iris = datasets.load_iris()
features = iris.data[:100,:]
target = iris.target[:100]
```

```
# 标准化特征
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# 创建一个逻辑回归的对象
logistic_regression = LogisticRegression(random_state=0)

# 训练模型
model = logistic_regression.fit(features_standardized, target)
```

讨论

逻辑回归是一种被广泛使用的二元分类器（也就是说，目标向量只能取两种值）。在逻辑回归中，线性模型（比如，$\beta_0 + \beta_1 x$）被包含在一个逻辑函数 $\frac{1}{1+e^{-z}}$（也叫作 sigmoid 函数）中。比如：

$$P(y_i = 1 \mid X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

其中，$P(y_i = 1 \mid X)$ 是第 i 个观察值的目标值 y_i 属于分类 1 的概率，X 是训练集的数据，β_0 和 β_1 是要学习的参数，e 是欧拉数（Euler's number）。逻辑函数的作用就是把函数的输出值限定在 0 到 1 之间，这样才能被解释为概率。如果 $P(y_i = 1 \mid X)$ 大于 0.5，那么 y_i 的预测分类为分类 1，否则就是分类 0。

在 scikit-learn 中，我们可以使用 `LogisticRegression` 学习一个逻辑回归模型。一旦被训练出来，这个模型就可以用于预测新观察值的分类：

```
# 创建一个新的观察值
new_observation = [[.5, .5, .5, .5]]

# 预测分类
model.predict(new_observation)

array([1])
```

在这个例子中，观察值被预测为分类 1。我们还可以进一步查看这个观察值属于各个分类的概率：

```
# 查看预测的概率
model.predict_proba(new_observation)

array([[ 0.18823041,  0.81176959]])
```

我们的观察值有 18.8% 的概率属于分类 0，81.1% 的概率属于分类 1。

16.2 训练多元分类器

问题描述

数据集中不止两个分类，需要训练出一个合适的分类器模型。

解决方案

在 scikit-learn 中，通过 LogisticRegression 使用一对多或者多项式方法来训练逻辑回归模型：

```
# 加载库
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 标准化特征
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# 创建一对多的逻辑回归对象
logistic_regression = LogisticRegression(random_state=0, multi_class="ovr")

# 训练模型
model = logistic_regression.fit(features_standardized, target)
```

讨论

独立地看，逻辑回归只是二元分类器，这意味着它不能处理多于两个分类的目标向量。但是，逻辑回归有两个巧妙的扩展可以解决这个问题。第一种扩展是一对多（One-vs-Rest，OVR）的逻辑回归，在这种逻辑回归中，对于每一个分类我们都会训练一个单独的模型来判断观察值是不是属于这个分类（这样就又变成二元分类问题了）。它有一个假设，即每一个分类问题（比如，观察值是否为分类 0）是相互独立的。

第二种扩展是多元逻辑回归（Multinomial Logistic Regression，MLR）。在 MLR 中，我

们在 15.1 节中看到的那个逻辑函数被一个 softmax 函数替换：

$$P(y_i = k \mid X) = \frac{e^{\beta_k x_i}}{\sum_{j=1}^{K} e^{\beta_j x_i}}$$

这里$P(y_i = k \mid X)$是第 i 个观察值的目标值 y_i 为类别 k 的概率，K 是分类的数量。MLR 有一个很实用的优点：它使用 `predict_proba` 方法预测概率，更可靠（即这个概率值被更好地校准过）。

使用 `LogisticRegression` 时，可以在这两种技术中选择，默认的选择是 OVR（`multi_class="ovr"`）。我们也可以把 `multi_class` 参数设置为 `multinomial`，改为使用 MLR。

16.3 通过正则化来减小方差

问题描述

减小逻辑回归模型的方差。

解决方案

调校正则化强度超参数 `C`：

```
# 加载库
from sklearn.linear_model import LogisticRegressionCV
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 标准化特征
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# 创建一个决策树分类器的对象
logistic_regression = LogisticRegressionCV(
    penalty='l2', Cs=10, random_state=0, n_jobs=-1)
```

```
# 训练模型
model = logistic_regression.fit(features_standardized, target)
```

讨论

正则化（regularization）是一种通过惩罚复杂模型来减小其方差的方法。准确地说，就是将一个罚项加在我们希望最小化的损失函数上，通常称为 L1 惩罚和 L2 惩罚。L1 惩罚是：

$$\alpha \sum_{j=1}^{p} \left|\hat{\beta}_j\right|$$

这里 $\hat{\beta}_j$ 是要学习的 p 个特征中的第 j 个所对应的参数，α 是一个超参数，表示正则化的强度。

L2 惩罚是：

$$\alpha \sum_{j=1}^{p} \hat{\beta}_j^2$$

α 的取值越大，对越大的参数值（也就是更复杂的模型）的惩罚就越重。scikit-learn 遵循常规用法，使用 C 来代替 α，这里的 C 等于正则化强度值的倒数，即 $C = \frac{1}{\alpha}$。

在使用逻辑回归的过程中，为了减小方差，我们把 C 当作需要被调校的超参数，以寻找一个可以创建最佳模型的 C 值。在 scikit-learn 中，可以使用 `LogisticRegressionCV` 类来有效地调校 C。`LogisticRegressionCV` 中的参数 `Cs` 可以接受两类值：一类是用来搜索的 C 的取值范围（如果提供了一个浮点数列表作为参数的话），另一类是一个整型数值。对于整型数值，`LogisticRegressionCV` 会生成一个列表，其长度为这个整型数值，里面的元素是从对数空间中取的大小在 −10,000 到 10,000 之间的值。

遗憾的是，`LogisticRegressionCV` 不允许我们比较不同的罚项。如果你想这么做，就必须使用第 12 章中提到的模型选择技术，但是它的效率会低一些。

16.4 在超大数据集上训练分类器

问题描述

在一个超大数据集上训练一个简单的分类器模型。

解决方案

在 scikit-learn 中通过 `LogisticRegression` 使用随机平均梯度（Stochastic Average Gradient，SAG）solver 来训练一个逻辑回归模型：

```
# 加载库
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 标准化特征
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# 创建逻辑回归对象
logistic_regression = LogisticRegression(random_state=0, solver="sag")

# 训练模型
model = logistic_regression.fit(features_standardized, target)
```

讨论

scikit-learn 的 `LogisticRegression` 提供了不少用于训练逻辑回归模型的方法，这些方法被称为 *solver*。大部分情况下，scikit-learn 会帮我们自动选择最佳 solver，或者警告我们该 solver 不能做某些事情。

虽然本书中不会给出详尽的解释（如果想了解更多信息，可以看下面“延伸阅读”中列出的 Mark Schmidt 的讲义第 264 页），但是随机平均梯度算法确实可以使我们在超大数据集上比使用其他 solver 更快地训练出一个模型。有一点需要注意，它对于特征的单位特别敏感，所以标准化特征这个步骤尤其重要。我们可以在代码中设置 `solver='sag'`，让学习算法使用这个 solver。

延伸阅读

- 《通过随机平均梯度算法来最小化有限和》（Mark Schmidt，*http://bit.ly/2GRrVw0*）

16.5 处理不均衡的分类

问题描述

训练一个简单的分类器模型。

解决方案

在 scikit-learn 中使用 LogisticRegression 来训练一个逻辑回归模型：

```
# 加载库
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 移除前 40 个观察值，使分类严重不均衡
features = features[40:,:]
target = target[40:]

# 创建目标向量，0 代表分类为 0，1 代表除分类 0 以外的其他分类
target = np.where((target == 0), 0, 1)

# 标准化特征
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# 创建决策树分类器对象
logistic_regression = LogisticRegression(random_state=0, class_
weight="balanced")

# 训练模型
model = logistic_regression.fit(features_standardized, target)
```

讨论

和 scikit-learn 中的很多其他学习算法一样，LogisticRegression 自带了一个处理不均衡分类的方法。如果数据集中的分类特别不均衡，而且在数据预处理过程中并没有解决这

个问题，就可以使用 class_weight 参数给分类设置权重，确保数据集中的各个分类是均衡的。具体地说，就是 balanced 参数值会自动给各分类加上权重，而权重值与分类出现频率的倒数相关：

$$w_j = \frac{n}{kn_j}$$

这里 w_j 是分类 j 的权重，n 是观察值的数量，n_j 是属于分类 j 的观察值的数量，k 是分类的总数。

第 17 章

支持向量机

17.0 简介

要理解支持向量机，必须先理解超平面。准确地说，一个超平面就是 n 维空间中的 $n-1$ 维空间。听起来很复杂，但其实很简单。打个比方，如果要分割一个二维空间，我们会使用一维的超平面（即一条线）。如果想分割一个三维空间，我们会使用一个二维的超平面(比如一张平整的纸或者一张床单)。超平面只不过是把这些概念应用到了 n 维空间中。

支持向量机通过寻找一个能最大化训练集数据中分类间距的超平面来给数据分类。以一个有两个分类的二维空间为例，我们可以把这个超平面想象成一条将两个分类隔开的最宽的笔直的“间隔带”（就是一条离两个分类都有一定间距的粗线）。

本章讲述在各种不同的情形下如何训练支持向量机，并进一步探讨怎样使用这个方法处理一些常见问题。

17.1 训练一个线性分类器

问题描述

训练一个模型对观察值分类。

解决方案

用支持向量分类器(Support Vector Classifier,SVC)来寻找最大化分类之间间距的超平面：

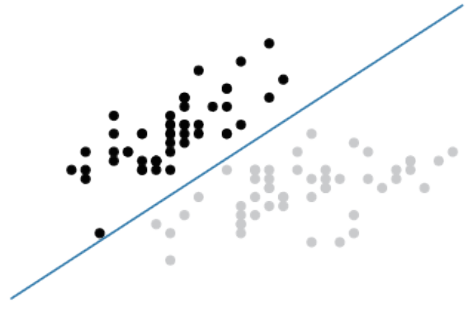
```
# 加载库
from sklearn.svm import LinearSVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# 加载数据，数据里只有两种分类和两个特征
iris = datasets.load_iris()
features = iris.data[:100,:2]
target = iris.target[:100]

# 标准化特征
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# 创建支持向量分类器
svc = LinearSVC(C=1.0)

# 训练模型
model = svc.fit(features_standardized, target)
```

讨论

scikit-learn 的 `LinearSVC` 实现了一个简单的 SVC。为了让你对 SVC 的作用有一个直观的认识，我们在下页的图中画出了样本点和超平面。尽管 SVC 在高维空间有很好的表现，但上述解决方案中只加载了两个特征和一部分样本数据，数据集内只有两个分类的数据，因此我们可以可视化这个模型。回忆一下前面的内容，SVC 一直试图找到一个能最大化分类之间间距的超平面，因为本例中只有两个维度，所以这里的超平面就是一条直线。在下面的代码中，我们把两个分类画在一个二维空间上，然后画出了这个超平面。

```
# 加载库
from matplotlib import pyplot as plt

# 画出样本点，并且根据其分类上色
color = ["black" if c == 0 else "lightgrey" for c in target]
plt.scatter(features_standardized[:,0], features_standardized[:,1], c=color)

# 创建超平面
w = svc.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(-2.5, 2.5)
yy = a * xx - (svc.intercept_[0]) / w[1]

# 画出超平面
```

```
plt.plot(xx, yy)
plt.axis("off"), plt.show();
```



在上图中，分类 0 的所有样本点都是黑色的，分类 1 的所有样本点都是浅灰色的。超平面就是决定新的样本点属于哪一个分类的分界线。也就是说，在这条线上方的所有样本点会被归为分类 0，而在这条线下方的样本点会被归为分类 1。我们可以看到，如果在左上角的空间创建一个新的样本点，那么它会被预测为属于分类 0：

```
# 创建一个新的样本点
new_observation = [[ -2,  3]]

# 预测新样本点的分类
svc.predict(new_observation)

array([0])
```

对于 SVC，有几点需要特别注意。首先，为了方便可视化，我们把本例限定为二元分类问题（只有两个分类），其实 SVC 在处理多元分类问题时表现也不错。其次，如上图所示，本例中的超平面被定义为直线（没有弯曲）。因为我们的数据集是线性可分的，也就是说存在一个超平面可以完美地分开这两个分类的样本点，所以在这个例子中超平面就是一条直线。但不幸的是，在真实世界中，这种情况并不多见。

更常见的情况是，我们并不能完美地将数据分类。在这些情况下，需要在 SVC 最大化超平面两侧的间距和最小化分类错误之间取得平衡。在 SVC 中，后者是通过一个超参数 *C*（分类错误时所需要接受的惩罚）来控制的。*C* 是 SVC 学习器的一个参数，也是学习器将一个样本点分类错误时被施加的罚项。当 *C* 很小时，SVC 可以容忍更多的样本点被错误分类（偏差大，方差小）。当 *C* 很大时，SVC 会因为对数据的错误分类而被重罚，因此通过反向传播来避免对样本点的错误分类（偏差小、方差大）。

在 scikit-learn 中，C 是由参数 `C` 所决定的，默认值是 `1.0`。应该把 C 当作机器学习算法中的一个超参数，可以用第 12 章中讲的模型选择技术进行调校。

17.2 使用核函数处理线性不可分的数据

问题描述

你要训练一个 SVC，但是你的数据是线性不可分的。

解决方案

使用核函数训练支持向量机的一个扩展，以创建非线性的决策边界：

```
# 加载库
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# 设置随机种子
np.random.seed(0)

# 生成两个特征
features = np.random.randn(200, 2)

# 使用异或门（你不需要知道原因）创建线性不可分的数据
target_xor = np.logical_xor(features[:, 0] > 0, features[:, 1] > 0)
target = np.where(target_xor, 0, 1)

# 创建一个有径向基核函数的支持向量机
svc = SVC(kernel="rbf", random_state=0, gamma=1, C=1)

# 训练分类器
model = svc.fit(features, target)
```

讨论

本书不会对支持向量机进行全面的解释，不过，简单了解一下支持向量机和核函数的背景知识还是有用的。为了便于理解，可以将 SVC 表示为：

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K\left(x_i, x_{i'}\right)$$

这里β_0是偏差，S是所有支持向量观察值的集合，α是要学习的模型参数，$(x_i, x_{i'})$是一对支持向量观察值。最重要的是核函数K，它会比较x_i和$x_{i'}$的相似度（如果你还不理解什么是核函数，不用着急）。对于K，只需要知道两点：第一，它决定了我们用什么类型的超平面来分离不同的分类；第二，我们使用不同的核函数来创建不同的超平面。比如，如果你想要17.1节中创建的基本线性超平面，就可以使用线性核函数：

$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij}x_{i'j}$$

这里p是特征的数量。如果我们想获取一个非线性决策边界，可以用多项式核函数来替换上面的线性核函数：

$$K(x_i, x_{i'}) = \left(1 + \sum_{j=1}^{d} x_{ij}x_{i'j}\right)^2$$

这里d是多项式核函数的度。我们还可以使用支持向量机中最通用的一种核函数，径向基核函数（radial basis function kernel）：

$$K(x_i, x_{i'}) = e^{\left(-\gamma\sum_{j=1}^{p}\left(x_{ij}x_{i'j}\right)^2\right)}$$

这里γ是一个超参数，而且必须大于0。前面的这些解释的要点在于，如果数据是线性不可分的，我们可以用其他可选的核函数来替换线性核函数，以创建一个非线性的超平面决策边界。

我们可视化一个简单的例子来理解核函数背后的逻辑。这个例子是基于Sebastian Raschka的工作设计的一个函数，它画出了观察值和一个二维空间里的超平面决策边界。你不需要理解这个函数是如何工作的。该函数的代码如下，你可以自己试一试：

```
# 画出观察值和超平面决策边界
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

def plot_decision_regions(X, y, classifier):
    cmap = ListedColormap(("red", "blue"))
    xx1, xx2 = np.meshgrid(np.arange(-3, 3, 0.02), np.arange(-3, 3, 0.02))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.1, cmap=cmap)
```

```
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker="+", label=cl)
```

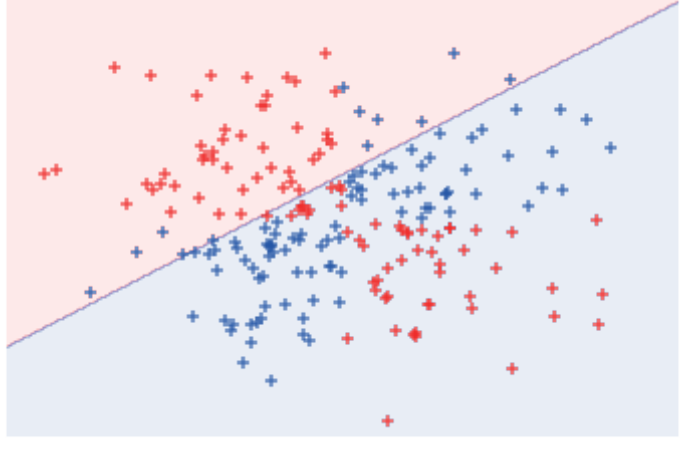
在本节的解决方案中，我们的数据有两个特征（即两个维度）和一个包含所有观察值分类的目标向量。重要的一点是，这些数据被设计为线性不可分的。换句话说就是，不存在一条直线可以完美地分开这两个分类的数据。首先，创建一个使用线性核函数的SVC：

```
# 创建一个使用线性核函数的 SVC
svc_linear = SVC(kernel="linear", random_state=0, C=1)

# 训练模型
svc_linear.fit(features, target)
```

```
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='linear',
  max_iter=-1, probability=False, random_state=0, shrinking=True,
  tol=0.001, verbose=False)
```

接下来，因为我们的数据只有两个特征，所以可以在二维空间内画出观察值、它们的分类和模型的线性超平面（见下图）：

```
# 画出观察值和超平面
plot_decision_regions(features, target, classifier=svc_linear)
plt.axis("off"), plt.show();
```

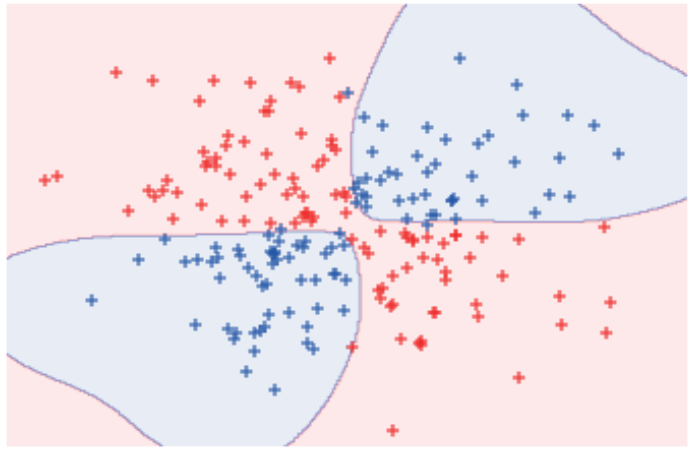


如上图所示，这个线性超平面的分类效果很差！让我们用一个径向基核函数来替换线性核函数，然后用它来训练一个新模型：

```
# 创建一个使用径向基核函数的 SVC
svc = SVC(kernel="rbf", random_state=0, gamma=1, C=1)

# 训练这个分类器
model = svc.fit(features, target)
```

然后我们画出观察值和超平面：

```
# 画出观察值和超平面
plot_decision_regions(features, target, classifier=svc)
plt.axis("off"), plt.show();
```



使用径向基核函数，可以创建一个分类效果比线性核函数好很多的决策边界。这也是在 SVC 中使用这些核函数的原因。

在 scikit-learn 中，我们可以通过设置 `kernel` 参数的值来选择想用的核函数。一旦选择了一个核函数，就需要为这个核函数确定一些合适的选项值，比如前面提到的多项式核函数中的 d（通过 `degree` 参数来设置）和径向基核函数中的 γ（通过 `gamma` 参数来设置）。我们还需要设置惩罚参数 `C`。在训练模型时，大部分情况下需要把这些参数视作超参数，然后用模型选择的技术找出能产生性能最佳的模型的参数值组合。

17.3 计算预测分类的概率

问题描述

想知道观察值被预测为某个分类的概率。

解决方案

如果使用 scikit-learn 的 SVC，可以设置 probability=True，然后训练模型，接着可以使用 predict_proba 来查看校准后的概率：

```
# 加载库
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 标准化特征
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# 创建 SVC 对象
svc = SVC(kernel="linear", probability=True, random_state=0)

# 训练分类器
model = svc.fit(features_standardized, target)

# 创建一个新的观察值
new_observation = [[.4, .4, .4, .4]]

# 查看观察值被预测为不同分类的概率
model.predict_proba(new_observation)

array([[ 0.00588822,  0.96874828,  0.0253635 ]])
```

讨论

前面讲过的很多监督学习算法都使用概率来预测分类。比如，在 KNN 算法中，观察值

的 k 个邻居的分类被记作投票数，以此计算观察值属于某个分类的概率。概率最大的分类就被认为是这个观察值所属的类别。SVC 算法使用一个超平面来创建决策区间，这种做法并不会直接计算出观察值属于某个分类的概率。但是，我们可以输出校准过的分类概率，并给出几点说明。在有两个分类的 SVC 中可以使用 Platt 缩放（Platt scaling），它首先训练这个 SVC，然后训练一个独立的交叉验证逻辑回归模型将 SVC 的输出转换为概率：

$$P(y = 1 \mid x) = \frac{1}{1 + e^{(A * f(x) + B)}}$$

这里 A 和 B 是参数向量，f 是第 i 个观察值到超平面的距离。如果数据集中不止两个分类，就可以使用 Platt 缩放的扩展。

计算预测分类的概率有两个主要的问题：第一，因为我们还训练了一个带交叉验证的模型，所以生成预测分类概率的过程会显著增加模型训练的时间；第二，因为预测的概率是通过交叉验证计算出来的，所以它们可能不会总是与预测的分类匹配。也就是说，一个观察值可能被预测为属于分类 1，但是它被预测为属于分类 1 的概率却小于 0.5。

在 scikit-learn 中，这些预测的概率必须在训练该模型时计算出来。我们可以通过设置 SVC 的 `probability` 参数为 `True` 来做到这一点。在模型被训练完之后，可以使用 `predict_proba` 方法输出观察值为每个分类的预测概率。

17.4 识别支持向量

问题描述

找出哪些观察值是决策超平面的支持向量。

解决方案

训练模型，然后使用 `support_vectors_` 方法：

```
# 加载库
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# 加载数据，数据中只有两个分类
iris = datasets.load_iris()
```

```
features = iris.data[:100,:]
target = iris.target[:100]

# 标准化特征
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# 创建 SVC 对象
svc = SVC(kernel="linear", random_state=0)

# 训练分类器
model = svc.fit(features_standardized, target)

# 查看支持向量
model.support_vectors_

array([[-0.5810659 ,  0.43490123, -0.80621461, -0.50581312],
       [-1.52079513, -1.67626978, -1.08374115, -0.8607697 ],
       [-0.89430898, -1.46515268,  0.30389157,  0.38157832],
       [-0.5810659 , -1.25403558,  0.09574666,  0.55905661]])
```

讨论

超平面是由相对而言的一小部分观察值（被称为支持向量）所决定的，支持向量机这个名字由此得来。直观地说，你可以把超平面想象成是被这些支持向量“举起来”的。因此，这些支持向量对于模型来说非常重要。比如，如果从数据集中移除了一个非支持向量的观察值，那么模型不会改变；但是如果移除一个支持向量，超平面与分类之间的间距就不会是最大的了。

训练完 SVC 之后，scikit-learn 提供了很多识别支持向量的选项。我们的解决方案用 `support_vectors_` 来输出模型中观察值特征的 4 个支持向量。我们也可以使用 `support_` 来查看支持向量在观察值中的索引值：

```
model.support_
array([23, 41, 57, 98], dtype=int32)
```

最后，我们可以使用 `n_support_` 来查看每个分类有几个支持向量：

```
model.n_support_
array([2, 2], dtype=int32)
```

17.5 处理不均衡的分类

问题描述

用不均衡的分类数据训练一个 SVC。

解决方案

使用 class_weight 来增加对数据量少的类别分错类后的惩罚：

```
# 加载库
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# 加载只有两个分类的数据
iris = datasets.load_iris()
features = iris.data[:100,:]
target = iris.target[:100]

# 删除前 40 个观察值，让各个分类的数据分布不均衡
features = features[40:,:]
target = target[40:]

# 创建目标向量，数值 0 代表分类 0，其他分类用数值 1 表示
target = np.where((target == 0), 0, 1)

# 标准化特征
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# 创建 SVC
svc = SVC(kernel="linear", class_weight="balanced", C=1.0, random_state=0)

# 训练分类器
model = svc.fit(features_standardized, target)
```

讨论

在支持向量机中，C 是一个超参数，它决定着一个观察值被分错类后的惩罚。在支持向量机中处理分类数据不均衡的一个方法是，对不同的分类使用不同的权重 C：

$$C_j = C^* w_j$$

这里 C 是对错误分类的惩罚，w_j 跟分类 j 出现的概率反相关，C_j 是分类 j 的 C 值。这背后大概的思路就是，增加对数据少的类别分错类时的惩罚，来防止模型被数据多的分类所“占据”。

在 scikit-learn 中使用 SVC 时，可以设置 `class_weight='balanced'` 自动为 C_j 取值。`balanced` 参数值自动为分类设置不同的权重：

$$w_j = \frac{n}{kn_j}$$

这里 w_j 是分类 j 的权重，n 是观察值的数量，n_j 是分类 j 的观察值的数量，k 是分类的数量。

第 18 章

朴素贝叶斯

18.0 简介

贝叶斯定理是已知事件 A 发生的先验概率 $P(A)$，以及事件 A 发生的条件下事件 B 发生的概率 $P(B \mid A)$ 时，获得概率 $P(A \mid B)$ 的重要方法：

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

贝叶斯定理受欢迎的程度在近十年飙升，学术界和商界都出现了很多挑战传统概率学的应用。在机器学习中，贝叶斯定理在分类上的应用之一就是朴素贝叶斯分类器。朴素贝叶斯分类器将实用机器学习中的一些优点整合到一个分类器中，这些优点包括：

1．方法直观。

2．对于小样本量的数据也能起作用。

3．进行训练和预测的计算成本低。

4．对于各种不同的参数设置总能得到稳定的结果。

具体来说，朴素贝叶斯分类器是基于下面的公式工作的：

$$P\left(y \mid x_1, \cdots, x_j\right) = \frac{P\left(x_1, \cdots, x_j \mid y\right)P(y)}{P\left(x_1, \cdots, x_j\right)}$$

这里：

- $P(y|x_1,...,x_j)$ 叫作后验概率，表示一个观察值在其 j 个特征是 $x_1,...,x_j$ 的情况下，它的分类是类别 y 的概率。
- $P(x_1,...,x_j|y)$ 叫作似然概率，表示给定观察值的分类 y，其特征是 $x_1,...,x_j$ 的概率。
- $P(y)$ 叫作先验概率，是我们在查看数据之前对于分类 y 出现的概率的猜测。
- $P(x_1,...,x_j)$ 叫作边缘概率。

在朴素贝叶斯中，我们对观察值每一个可能的分类的后验概率进行比较。具体来说，因为在这些比较中边缘概率是恒定不变的，所以只需要比较每个分类的后验概率就好。对于每一个观察值，后验概率最大的分类就是这个观察值的预测分类。

对于朴素贝叶斯分类器来说，有两个重要的地方需要特别指出。第一，对于数据的每个特征，必须假定它的似然概率 $P(x_j \mid y)$ 的统计学分布。最常用的分布有正态（高斯）分布、多项式分布和伯努利分布。对这些分布的选择总是由特征的特性（比如连续、二分等）决定的。第二，朴素贝叶斯得名于一个假设——每个特征和它的似然概率是相互独立的。这种“朴素”的假设经常是错的，但是在实际操作中它并不影响我们构建一个高品质的分类器。

在本章中，我们会讲解使用 scikit-learn 基于 3 种不同的似然分布训练 3 种类型的朴素贝叶斯分类器。

18.1 为连续的数据训练分类器

问题描述

数据的特征是连续的（非离散值），我们想训练一个朴素贝叶斯分类器。

解决方案

在 scikit-learn 中使用高斯朴素贝叶斯分类器：

```
# 加载库
from sklearn import datasets
from sklearn.naive_bayes import GaussianNB

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建高斯朴素贝叶斯对象
```

```
classifer = GaussianNB()

# 训练模型
model = classifer.fit(features, target)
```

讨论

最常用的朴素贝叶斯分类器就是高斯朴素贝叶斯分类器。在这个分类器中，我们假设对于一个分类为 y 的观察值，其特征 x 的似然概率服从正态分布：

$$p(x_j \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} e^{-\frac{(x_j - \mu_y)^2}{2\sigma_y^2}}$$

这里 σ_y^2 和 μ_y 是分类 y 的特征 x_j 的方差和平均值。因为我们假设观察值的特征服从正态分布，所以在所有特征都是连续的数的场景下，最适合使用高斯朴素贝叶斯分类器。

在 scikit-learn 中，我们训练高斯朴素贝叶斯分类器时像训练其他模型一样使用了 fit 方法，然后就可以预测一个观察值的分类了：

```
# 创建一个观察值
new_observation = [[ 4, 4, 4, 0.4]]

# 预测分类
model.predict(new_observation)

array([1])
```

朴素贝叶斯分类器有一点很有意思，它允许我们为每个目标分类设定一个先验概率。可以使用 GaussianNB 的 priors 参数来设定，这个参数的值是一个列表，其中的元素为目标向量中每个分类的概率：

```
# 给定每个分类的先验概率，创建一个高斯朴素贝叶斯对象
clf = GaussianNB(priors=[0.25, 0.25, 0.5])

# 训练模型
model = classifer.fit(features, target)
```

如果不给 priors 参数设定任何值，就会基于我们的数据来计算先验概率。

最后需要指出一点，GaussianNB 产生的预测概率（使用 predict_proba 输出）是没有经过校正的。也就是说，它们是不可信的。如果想生成有用的预测概率，就需要用保序回

归（isotonic regression）或者其他相关方法来校正。

延伸阅读

- 《在机器学习中朴素贝叶斯分类器是如何工作的》（Dataaspirant，*http:// bit.ly/2F6trtt*）

18.2 为离散数据和计数数据训练分类器

问题描述

给定离散数据或者计数数据（count data），训练一个朴素贝叶斯分类器。

解决方案

使用多项式朴素贝叶斯分类器：

```
# 加载库
import numpy as np
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer

# 创建文本
text_data = np.array(['I love Brazil. Brazil!',
                      'Brazil is best',
                      'Germany beats both'])

# 创建词袋
count = CountVectorizer()
bag_of_words = count.fit_transform(text_data)

# 创建特征矩阵
features = bag_of_words.toarray()

# 创建目标向量
target = np.array([0,0,1])

# 给定每个分类的先验概率，创建一个多项式朴素贝叶斯对象
classifer = MultinomialNB(class_prior=[0.25, 0.5])

# 训练模型
model = classifer.fit(features, target)
```

讨论

多项式朴素贝叶斯分类器与高斯朴素贝叶斯分类器的工作原理类似，但是样本数据的特征为服从多项式分布。在实际应用中，这意味着这种分类器主要用于数据是离散的情况（比如对电影的评分，从 1 到 5）。最常见的使用多项式朴素贝叶斯分类器的场景是文本分类，使用词袋或者 TF-IDF 方法（参见 6.8 和 6.9 节）。

在本节的解决方案中，我们创建了一个玩具文本数据集，里面有 3 个观察值（3 句话），然后把文本字符串转换为词袋特征矩阵和一个目标向量。接着，我们使用 `MultinomialNB` 来训练一个模型，并给定两个分类（支持巴西队和支持德国队）的先验概率。

`MultinomialNB` 和 `GaussianNB` 的工作原理相似，模型都是用 `fit` 方法来训练的，观察值可以用 `predict` 来预测分类：

```
# 创建一个观察值
new_observation = [[0, 0, 0, 1, 0, 1, 0]]

# 预测新观察值的分类
model.predict(new_observation)

array([0])
```

如果没有指定 `class_prior` 的值，会从数据中学习而得到先验概率。但是如果想要用均匀分布计算先验概率的话，可以设置 `fit_prior=False`。

最后，`MultinomialNB` 含有一个附加的平滑超参数 `alpha`，它需要被调校。它的默认值是 `1.0`，设置为 `0.0` 就意味着不做平滑。

18.3 为具有二元特征的数据训练朴素贝叶斯分类器

问题描述

你有一些具有二元特征的数据，需要训练一个朴素贝叶斯分类器。

解决方案

使用伯努利朴素贝叶斯分类器：

```
# 加载库
import numpy as np
from sklearn.naive_bayes import BernoulliNB

# 创建三个二元特征
features = np.random.randint(2, size=(100, 3))

# 创建一个二元目标向量
target = np.random.randint(2, size=(100, 1)).ravel()

# 给定每个分类的先验概率，创建一个多项式伯努利朴素贝叶斯对象
classifer = BernoulliNB(class_prior=[0.25, 0.5])

# 训练模型
model = classifer.fit(features, target)
```

讨论

伯努利朴素贝叶斯分类器假设所有的特征都是二元分类的，这样它们就只有两种值（比如，经过 one-hot 编码的名义上的分类特征）。像它的多项式“表亲”一样，伯努利朴素贝叶斯经常被用于文本分类，这时特征矩阵表示的是某个词是否在一个文档中出现了。和 `MultinomialNB` 一样，`BernoulliNB` 还有一个平滑超参数 `alpha`，我们会使用模型选择技术来调校它。最后，如果想用先验概率的话，可以使用 `class_prior` 参数，它带有一个列表，里面包含每个分类的先验概率值。如果想指定统一的先验概率，可以设置 `fit_prior=False`：

```
model_uniform_prior = BernoulliNB(class_prior=None, fit_prior=False)
```

18.4 校准预测概率

问题描述

校准由朴素贝叶斯分类器得出的预测概率，使它们可以被解释。

解决方案

使用 `CalibratedClassifierCV`：

```
# 加载库
from sklearn import datasets
from sklearn.naive_bayes import GaussianNB
from sklearn.calibration import CalibratedClassifierCV
```

```
# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target

# 创建高斯朴素贝叶斯对象
classifer = GaussianNB()

# 创建使用 sigmoid 校准调校过的交叉验证模型
classifer_sigmoid = CalibratedClassifierCV(classifer, cv=2, method='sigmoid')

# 校准概率
classifer_sigmoid.fit(features, target)

# 创建新的观察值
new_observation = [[ 2.6,  2.6,  2.6,  0.4]]

# 查看校准过的概率
classifer_sigmoid.predict_proba(new_observation)

array([[ 0.31859969,  0.63663466,  0.04476565]])
```

讨论

分类的概率是机器学习模型中常见且有用的组成部分。在 scikit-learn 中，大部分学习算法允许我们使用 `predict_proba` 查看各个分类的预测概率。这一点极其有用。比如，模型预测的结果是观察值有超过 90% 的概率属于某一个分类，而我们只想预测为一个特定分类。但是，包括朴素贝叶斯分类器在内的一些模型，它们输出的概率并不是真实世界的概率。换句话说，`predict_proba` 有可能预测一个观察值属于某一个分类的概率为 `0.7`（70%），但是在真实世界中，这个概率可能是 10% 或者 99%。特别是在朴素贝叶斯中，尽管对不同目标分类的预测概率的排序是正确的，但是原始的预测概率往往为极端的值，趋近于 0 或者 1。

为了获得有意义的预测概率，我们可以使用校准（*calibration*）方法。在 scikit-learn 中，可以使用 `CalibratedClassifierCV` 类，创建经由 *k* 折交叉验证调校过的预测概率。在 `CalibratedClassifierCV` 中，训练集被用来训练模型，测试集被用来校准预测概率，返回的预测概率是 *k* 折的平均值。

使用我们的解决方案，可以看到原始预测概率和调校过的预测概率之间的差别。在解决方案中，我们创建了一个高斯朴素贝叶斯分类器。如果训练这个分类器后，用它预测一个新观察值的分类概率，就会看到很极端的概率预测值：

```
# 训练一个高斯朴素贝叶斯分类器来预测观察值的分类概率
classifer.fit(features, target).predict_proba(new_observation)

array([[  2.58229098e-04,   9.99741447e-01,   3.23523643e-07]])
```

但是，如果我们在这之后校准预测概率（在解决方案中就是这样做的），就会得到很不一样的结果：

```
# 查看校准过的概率
classifer_sigmoid.predict_proba(new_observation)

array([[ 0.31859969,  0.63663466,  0.04476565]])
```

`CalibratedClassifierCV` 提供了两种校准方法：Platt sigmoid 模型和保序回归，可以通过 `method` 参数来设置。受篇幅所限，这里不讲解其中的细节，但是需要说明一点，保序回归是无参的，在样本量很小的情况下（比如，只有 100 个观察值），它往往会过拟合。在上述解决方案中，因为所用的鸢尾花数据集只有 150 个观察值，所以我们使用了 Platt sigmoid 模型。

第 19 章

聚类

19.0 简介

本书的大部分篇幅都在讨论有监督学习，这意味着我们可以获取特征和目标数据。可惜的是，在现实世界中不总是这样，我们经常会遇到只知道特征的场景。举个例子，假设我们手头有一家百货商店的销售数据，现在要把这些数据按照购物者是否为折扣俱乐部的会员分成两类。在这个例子中不可能使用有监督学习，因为我们并没有一个用于训练和评估模型的目标。尽管如此，我们还有另一个选择：无监督学习。如果折扣俱乐部的会员和非会员在百货商店中的行为是完全不同的，那么两个会员行为上的平均差异会比会员和非会员之间的平均差异小。也就是说，观察值有两个分类（cluster）[2]。

聚类算法的目标是找出这些观察值潜在的分类，如果做得好的话，我们能在没有目标向量的情况下预测观察值的分类。聚类算法有很多，它们使用了多种不同的方法来识别数据中的聚类。在本章中，我们会教大家使用 scikit-learn 实现一些聚类算法，并将其应用到实践中。

19.1 使用 K-Means 聚类算法

问题描述

要把观察值聚类为 k 个分类。

2 编者注：本章中“分组”“分类”“聚类”几个词含义是相同的，其实指的都是 cluster。考虑到语言的流畅性和易读性，并未全部统一表述为“聚类”。

解决方案

使用 K-Means 聚类算法：

```
# 加载库
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

# 加载数据
iris = datasets.load_iris()
features = iris.data

# 标准化特征
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# 创建 K-Means 对象
cluster = KMeans(n_clusters=3, random_state=0, n_jobs=-1)

# 训练模型
model = cluster.fit(features_std)
```

讨论

K-Means 聚类是最常见的一种聚类算法。在 K-Means 聚类中，算法试图把观察值分到 k 个组中，每个组的方差都差不多。分组的数量 k 是用户设置的一个超参数。具体来讲，K-Means 算法有如下几个步骤：

1. 随机创建 k 个分组（即 cluster）的“中心”点。
2. 对于每个观察值：
   （1）算出每个观察值和这 k 个中心点之间的距离。
   （2）将观察值指派到离它最近的中心点的分组。
3. 将中心点移动到相应分组的点的平均值位置。
4. 重复步骤 2 和 3，直到没有观察值需要改变它的分组。这时该算法就被认为已经收敛，而且可以停止了。

关于 K-Means 算法，有 3 点值得注意。第一，K-Means 聚类假设所有的聚类是凸形的（比如，圆形或者球形）。第二，所有特征在同一度量范围内。在解决方案中，我们对特征进行了标准化，以满足这个条件。第三，分组之间是均衡的（即每个分组中观察值的数

量大致相等）。如果你认为自己无法满足这些假设，就可能需要尝试其他的聚类方法。

在 scikit-learn 中，K-Means 聚类是通过 KMeans 类来实现的。其中最重要的参数就是 n_clusters，它设定聚类的数量 k。在有些情况下，数据本身的特性可以决定 k 的值（例如，学校中学生的数据就是一个年级为一个聚类），但是我们往往并不知道聚类的数量。在这些情况下，我们希望可以基于一些条件来选择 k 值。例如，轮廓系数（silhouette coefficient，参见 11.9 节）衡量的是同类中的相似度和不同类之间的相似度的比值。而且因为 K-Means 算法的计算开销很大，所以可能要使用计算机所有的内核，我们可以设置 n_jobs = -1 来实现这一点。

在解决方案中，我们使用的是鸢尾花数据集，但是耍了一点小花招：我们知道这个数据集有 3 个分类，因此设置 $k = 3$。用 labels_ 方法可以查看每个观察值的预测分类：

```
# 查看预测分类
model.labels_

array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 0, 0, 0, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2,
       2, 0, 2, 2, 2, 2, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 0, 0, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 2, 2,
       0, 0, 0, 0, 2, 0, 2, 0, 2, 0, 0, 2, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0,
       2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 2], dtype=int32)
```

如果拿这个预测分类跟观察值的真实分类进行比较，就会发现 K-Means 做得还不错：

```
# 查看真实分类
iris.target

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

但是正如你想的，如果分类的数量设置错了，那么 K-Means 的表现就会急剧下降。

最后，像 scikit-learns 的其他方法一样，我们可以使用训练好的分类来预测新观察值的分类：

```
# 创建新观察值
new_observation = [[0.8, 0.8, 0.8, 0.8]]
```

```
# 预测观察值的分类
model.predict(new_observation)

array([0], dtype=int32)
```

这个观察值被预测为离某个分类的中心点距离最近的分类。可以使用 cluster_centers_ 来查看这些中心点：

```
# 查看分类的中心点
model.cluster_centers_

array([[ 1.13597027, 0.09659843, 0.996271 , 1.01717187],
       [-1.01457897, 0.84230679, -1.30487835, -1.25512862],
       [-0.05021989, -0.88029181, 0.34753171, 0.28206327]])
```

延伸阅读

- 《K-Means 聚类简介》（DataScience.com，*http://bit.ly/2Hjik1f*）

19.2 加速 K-Means 聚类

问题描述

要把观察值分成 *k* 个组，但是用 K-Means 算法需要太长的时间。

解决方案

使用 Mini-Batch K-Means：

```
# 加载库
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import MiniBatchKMeans

# 加载数据
iris = datasets.load_iris()
features = iris.data

# 标准化特征
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# 创建 K-Means 对象
```

```
cluster = MiniBatchKMeans(n_clusters=3, random_state=0, batch_size=100)

# 训练模型
model = cluster.fit(features_std)
```

讨论

Mini-Batch K-Means 和我们在 19.1 节讨论的 K-Means 算法的工作原理类似。如果不深究过多细节，那么这两种算法的区别就是，前者计算量最大的步骤只在观察值的一部分随机样本上而非所有的观察值上执行。这个方法可以在只损失一小部分质量的情况下显著缩短算法收敛的时间。

`MiniBatchKMeans` 的用法和 `KMeans` 十分相似，最大的区别在于 `batch_size` 参数。`batch_size` 控制每个批次中随机选择的观察值的数量。批次中的观察值越多，在训练过程中需要花费的算力就越大。

19.3 使用 Meanshift 聚类算法

问题描述

在不对分类的数量和形状做假设的情况下对观察值聚类。

解决方案

使用 `MeanShift` 聚类：

```
# 加载库
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import MeanShift

# 加载数据
iris = datasets.load_iris()
features = iris.data

# 标准化特征
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# 创建 MeanShift 对象
cluster = MeanShift(n_jobs=-1)
```

```
# 训练模型
model = cluster.fit(features_std)
```

讨论

我们前面讨论的 K-Means 算法有一个缺点，这就是要在训练之前设定聚类的数量 *k*，而且还要假设聚类的形状。Meanshift 算法就没有这些限制。

Meanshift 是一个简单的概念，但不太好解释，用类比的方法来解释或许比较容易理解。想象有一个有雾气弥漫的足球场（即一个二维的特征空间），上面站着 100 个人（即我们的观察值）。因为雾很大，人只能看到很近的地方。每分钟每个人向四周看一看，然后朝着可以看到最多人的方向移动一步。随着时间流逝，因为人们一次次地朝着越来越大的人群移动，球场上的人开始聚集成一个个小组，最终这些人就在球场上形成了聚类（cluster）。每个人的分类被指定为他们最终所在的聚类。

在 scikit-learn 中实现 Meanshift 算法的是 `MeanShift`，它比上面的类比更复杂但遵循的逻辑是相同的。`MeanShift` 有两个重要的参数。第一个是 `bandwidth`，它设定了一个观察值用以决定移动方向的区域（又叫作核）的半径。在我们的类比中，`bandwidth` 就是一个人能在雾里能看到的距离。我们可以手动设定这个参数，但是默认情况下，`MeanShift` 会自动估计一个合理的 `bandwidth` 值（会显著增加计算开销）。第二，有时候执行 Meanshift 算法时，在一个观察值的核中看不到任何其他观察值。这就相当于球场上有一个人看不到任何其他人。默认情况下，`MeanShift` 把所有的这些“孤儿”观察值分配给离它最近的观察值的核。如果你想丢弃这些孤值，可以设置 `cluster_all=False`，这样所有孤值的标签就被设定为 `-1`。

延伸阅读

- Meanshift 聚类算法（EFAVDB，*http://efavdb.com/mean-shift/*）

19.4 使用 DBSCAN 聚类算法

问题描述

要把观察值分组成高密度的聚类。

解决方案

使用 DBSCAN 聚类算法：

```
# 加载库
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN

# 加载数据
iris = datasets.load_iris()
features = iris.data

# 标准化特征
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# 创建 DBSCAN 对象
cluster = DBSCAN(n_jobs=-1)

# 训练模型
model = cluster.fit(features_std)
```

讨论

聚类是很多观察值紧密聚集在一起的区域，DBSCAN 算法就是受这一点的启发而来的，它对于聚类的形状没有做任何假设。具体来说，DBSCAN 算法有如下几步：

1. 先选择一个随机的观察值。
2. 如果 x_i 的近邻数为最小限度数量的话，就把它归入一个聚类。
3. 对 x_i 的所有邻居重复执行步骤 2，对邻居的邻居也如此，以此类推。这些点就是聚类的核心观察值。
4. 一旦步骤 3 处理完所有邻近的观察值，就选择一个新的随机点（重新开始执行步骤 1）。

一旦完成这些步骤，我们就会得到一个聚类的核心观察值的集合。最后，凡是在聚类附近但又不是核心的观察值将被认为属于这个聚类，而那些离聚类很远的观察值将被标记为噪声。

DBSCAN 对象需要设置以下 3 个主要参数。

eps

从一个观察值到另一个观察值的最远距离，超过这个距离将不再认为二者是邻居。

`min_samples`

最小限度的邻居数量，如果一个观察值在其周围小于 `eps` 距离的范围内有超过这个数量的邻居，就被认为是核心观察值。

`metric`

`eps` 所用的距离度量，比如 `minkowski`（闵可夫斯基距离）或者 `euclidean`（欧氏距离）。注意，如果使用闵可夫斯基距离，就可以用参数 `p` 设定闵可夫斯基距离中的幂次。

如果观察训练集数据中的聚类，可以看到有两个聚类被识别出来，被标记为 `0` 和 `1`，噪声观察值则被标记为 `-1`：

```
# 显示聚类的情况
model.labels_
```

```
array([  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, 0,
         0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -1,
         0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
         1, 1, 1, 1, 1, -1, -1, 1, -1, -1, 1, -1, 1, 1, 1, 1, 1,
        -1, 1, 1, 1, -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        -1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, -1, 1,
         1, 1, 1, -1, -1, -1, -1, -1, 1, 1, 1, 1, -1, 1, 1, -1, -1,
        -1, 1, 1, -1, 1, 1, -1, 1, 1, 1, -1, -1, -1, 1, 1, 1, -1,
        -1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1])
```

延伸阅读

- DBSCAN（Wikipedia，*https://en.wikipedia.org/wiki/DBSCAN*）

19.5 使用层次合并聚类算法

问题描述

使用聚类的层次来给观察值分组。

解决方案

使用 `Agglomerative` 聚类：

```
# 加载库
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.cluster import AgglomerativeClustering

# 加载数据
iris = datasets.load_iris()
features = iris.data

# 标准化特征
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# 创建一个 Agglomerative 聚类对象
cluster = AgglomerativeClustering(n_clusters=3)

# 训练模型
model = cluster.fit(features_std)
```

讨论

Agglomerative 聚类是一个强大的、灵活的层次聚类算法。在 Agglomerative 聚类中，所有观察值一开始都是一个独立的聚类。接着，满足一定条件的聚类被合并。不断重复这个合并过程，让聚类不断增长，直到达到某个临界点。在 scikit-learn 中，AgglomerativeClustering 使用 linkage 参数来决定合并策略，使其可以最小化下面的值：

1．合并后的聚类的方差（ward）。

2．两个聚类之间观察值的平均距离（average）。

3．两个聚类之间观察值的最大距离（complete）。

还有两个参数也很有用。第一个是 affinity，它决定 linkage 使用何种距离度量（比如 minkowski 或者 euclidean 等）。第二个是 n_clusters，它设定了聚类算法试图寻找的聚类的数量。也就是说，直到有 n_clusters 个聚类时，聚类的合并才结束。

和我们讨论的其他聚类算法一样，可以用 labels_ 方法来查看每个观察值所属的聚类：

```
# 显示聚类的情况
model.labels_
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
       1, 1, 1, 1, 0, 0, 0, 2, 0, 2, 0, 2, 0, 2, 2, 0, 2, 0, 2, 0, 2, 2, 2,
       2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 0, 2, 0, 0, 2, 2, 2, 2, 0,
       2, 2, 2, 2, 2, 0, 2, 2, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

第 20 章

神经网络

20.0 简介

神经网络的核心是*神经元*（*unit*）。神经元接收一个或者多个输入，为每个输入乘以一个参数（又叫作*权重*），接着对加权之后的输入值求和再加上某个偏差值（一般是 1），最后把这个值反馈给一个激活函数。这个输出值会被向前传递给神经网络中更深层的神经元（如果还有更深层的神经元的话）。

前馈（*feedforward*）神经网络，又叫作*多层感知器*（*multilayer perceptron*），是现实世界中最简单的人工神经网络。神经网络可以视为由一系列相互连接的层组成的网络，它的一端连接着一个观察值的特征值，另一端连接着对应的目标值（比如观察值的分类）。观察值的特征值在网络中向前传播，每经过一层，网络都会对特征值进行转换，目标是让最后的输出与目标值相同，“前馈”这个名字就源于此。

具体来说，前馈神经网络包含 3 种类型的神经元层。神经网络的起始端有一个输入层，输入层的每一个神经元包含一个观察值的某一个特征值。比如，如果一个观察值有 100 个特征，那么输入层就有 100 个节点。神经网络的末端有一个输出层，它把隐藏层的输出转换成对我们的任务有用的值。举个例子，如果我们的目标是对数据做二元分类，我们可以使用一个只有一个节点的输出层，这个节点使用 sigmoid 函数将它的输出限定在 0 和 1 之间，表示对观察值的预测分类的概率。夹在输入层和输出层之间的就是我们前面说的“隐藏”层（其实它们根本没有隐藏）。这些隐藏层接连不断地将输入层的特征值转换为一些值，这些值只要经输出层处理，就可以得到对应观察值的目标分类。有多个隐藏层的神经网络（比如 10 层、100 层、1000 层）被认为是很“深”的网络，它们的应用又被称为*深度学习*（*deep learning*）。

一般而言，神经网络在创建时所有的参数都会被初始化为符合高斯分布或者正态分布

的小随机值。一旦一个观察值（更常见的情况是一组观察值的集合，又被称为批次<*batch*>）被传给神经网络，它的输出值就会被拿来用损失函数与观察值的真实值进行比较。这个过程叫作前向传播（*forward propagation*）。接下来，算法“向后”在神经网络中传播，它识别出每个参数对预测值和真实值之间的差异的影响程度，这个过程叫作反向传播（*backward propagation*）。对于每一个参数，优化算法决定应该如何调整权重值才能改善输出值。

神经网络通过对训练集中的每个观察值重复做多次前向传播和反向传播（所有的观察值都通过网络传递一次就被称为一个 *epoch*，这种训练一般都需要多个 epoch），以此迭代更新参数的值。

在本章中，我们将使用流行的 Python 包 Keras 来创建、训练和评估几种神经网络。Keras 是一个高层的软件包，使用了像 TensorFlow 和 Theano 这样的库来作为它的“引擎”。对于我们来说，Keras 的优势就在于它可以使我们专注于神经网络的设计和训练，把张量运算的细节留给其他库来完成。

使用 Keras 创建的神经网络既可以用 CPU（比如，你的笔记本电脑）训练，也可以用 GPU（比如，某些专门为深度学习而设计的计算机）训练。在现实世界中，我们强烈建议使用 GPU 来训练神经网络；但是为了便于学习，本书中所有的神经网络都很小而且很简单，用笔记本电脑也可以在几分钟之内把它们训练出来。你只需要记住，对于大型神经网络和大量训练数据，使用 CPU 来训练会比使用 GPU 慢得多。

20.1 为神经网络预处理数据

问题描述

对数据进行预处理，使其可以用于神经网络。

解决方案

使用 scikit-learn 的 `StandardScaler` 标准化每个特征：

```
# 加载库
from sklearn import preprocessing
import numpy as np

# 创建特征
features = np.array([[-100.1, 3240.1],
```

```
                        [-200.2, -234.1],
                        [5000.5, 150.1],
                          [6000.6, -125.1],
                          [9000.9, -673.1]])

# 创建 scaler
scaler = preprocessing.StandardScaler()

# 转换特征
features_standardized = scaler.fit_transform(features)

# 展示特征
features_standardized

array([[-1.12541308, 1.96429418],
       [-1.15329466, -0.50068741],
       [ 0.29529406, -0.22809346],
       [ 0.57385917, -0.42335076],
       [ 1.40955451, -0.81216255]])
```

讨论

本节的内容和 4.3 节很相似，但因为它对于学习神经网络非常重要，所以我们在此复习一遍。一般来说，一个神经网络的参数会被初始化（或者说被创建）为一些小的随机数。如果特征值比参数值大很多，神经网络往往表现得不如人意。另外，观察值的特征值经过这些神经元的传递后，会进行相加，所以让所有的特征值拥有同样的单位就很重要。

由于上面这些原因，最佳实践（尽管有些时候并非必须如此，比如当我们的数据都是二元特征的时候）就是先标准化每一个特征值，使其均值为 0，标准差为 1。使用 scikit-learn 的 `StandardScaler`，很容易实现这一步。

可以查看我们的第一个特征的均值和标准差来看标准化的效果：

```
# 打印均值和标准差
print("Mean:", round(features_standardized[:,0].mean()))
print('"Standard deviation:", features_standardized[:,0].std())

Mean: 0.0
Standard deviation: 1.0
```

20.2 设计一个神经网络

问题描述

设计一个神经网络。

解决方案

使用 Keras 的 Sequential 模型：

```
# 加载库
from keras import models
from keras import layers

# 启动神经网络
network = models.Sequential()

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu", input_shape=(10,)))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu"))

# 添加使用 sigmoid 激活函数的全连接层
network.add(layers.Dense(units=1, activation="sigmoid"))

# 编译神经网络
network.compile(loss="binary_crossentropy", # 交叉熵
                optimizer="rmsprop", # 均方根传播
                metrics=["accuracy"]) # 将准确率作为性能指标
```

```
Using TensorFlow backend.
```

讨论

神经网络是由多层神经元组成的，神经元层的类型以及它们组成神经网络的方式非常多。现阶段尽管有很多被广泛使用的架构模式（我们会在本章讲到），但真相却是，选择正确的架构更像是一门艺术，需要我们做很多研究工作。

要在 Keras 中构建一个前馈神经网络，我们需要对网络架构和训练过程做许多选择。记住，隐藏层的每个神经元都会经历如下步骤：

1. 接收一些输入。

2．给每个输入乘以一个参数作为权重。

3．对所有加权过的输入求和，再加上偏差（一般是 1）。

4．接下来，将这个值应用到某个函数上（又叫作*激活函数*）。

5．把输出传递给下一层的神经元。

第一，对隐藏层和输出层中的每一层，我们都必须定义神经元的数量和它的激活函数。总的来说，一个层中神经元越多，神经网络就越能学习复杂的模式。尽管如此，神经元过多可能会使神经网络对训练数据过拟合，影响其在测试数据上的表现。

对隐藏层来说，一个流行的激活函数是*矫正的线性单元*（Rectified Linear Unit，ReLU）：

$$f(z) = \max(0, z)$$

这里 z 是加权过的输入和偏差之和。我们可以看到，如果 z 大于 0，激活函数就返回 z 值；否则返回 0。这个简单的激活函数有许多可贵的特性（本书不讨论这些特性），使它成为神经网络中受欢迎的激活函数。不过，你也要知道还存在很多其他的激活函数。

第二，我们需要决定神经网络中隐藏层的数量。层数越多，神经网络能学习的关系就越复杂，但是计算开销也会越大。

第三，我们必须决定输出层的激活函数（如果有的话）的结构。输出函数的本质经常由神经网络的目标决定。这里列出一些常见的输出层的模式：

二元分类

一个有 sigmoid 激活函数的神经元。

多元分类

k 个神经元（这里 k 是目标分类的个数）和一个 softmax 激活函数。

回归

一个没有激活函数的神经元。

第四，我们需要定义一个损失函数（用来衡量预测值和真实值的符合程度）。这个函数也经常是由问题的类型所决定的：

二元分类

二分交叉熵（Binary Cross-entropy）。

多元分类

分类交叉熵（Categorical cross-entropy）。

回归

均方误差。

第五，我们需要定义一个优化器，它可以被直观地理解为我们的策略“绕过了”损失函数，并且找到了产生最小误差的那些参数值。常用的优化器有：随机梯度下降、动量随机梯度下降、均方根传播和自适应矩估计（关于这些优化器的更多信息，可以查看本节末尾的“延伸阅读”中列出的资料）。

第六，我们可以选择一个或者多个指标来评估神经网络的性能，比如准确率。

Keras 提供了两种创建神经网络的方法。Keras 的 sequential 模型通过把神经元层堆叠起来以创建神经网络。另一种创建神经网络的方法叫作函数式 API，但是这种方法更适合研究人员使用而不适合商用。

在解决方案中，我们使用 Keras 的 sequential 模型创建了一个两层的神经网络（计算层数的时候，我们不会把输入层算在内，因为它没有任何参数需要学习）。每一层都是“紧密的”（又叫作全连接的），这意味着前一层的所有神经元都和下一层的所有神经元相连。在第一个隐藏层中，我们设定 `units = 16`，表示这一层有 16 个神经元，每个神经元都有 ReLU 激活函数（`activation='relu'`）。在 Keras 中，任何神经网络的第一个隐藏层都必须包含一个 `input_shape` 参数，它表示特征数据的形状。比如，`(10,)` 就告诉我们，第一层期望每个观察值都有 10 个特征值。第二层和第一层一样，只不过不需要加上 `input_shape` 参数。我们的神经网络是设计来做二元分类的，所以输出层仅包含一个带 sigmoid 激活函数的神经元，它将输出限制在 0 和 1 之间（表示一个观察值属于分类 1 的概率）。

最后，在训练模型之前，还需要告诉 Keras，我们想让网络如何学习。我们用 `compile` 方法，加上优化算法（`RMSProp`）、损失函数（`binary_crossentropy`），以及一个或者多个性能衡量标准来告诉 Keras 如何学习。

延伸阅读

- Keras 文档：损失函数（*https://keras.io/losses/*）
- 分类的损失函数（Wikipedia，*http://bit.ly/2FwpCkM*）
- 《分类问题中深度神经网络的损失函数》（Katarzyna Jano cha, Wojciech Marian Czarnecki，*https://arxiv.org/abs/1702.05659*）

20.3 训练一个二元分类器

问题描述

训练一个二元分类的神经网络。

解决方案

使用 Keras 构建一个前馈神经网络，然后用 fit 方法训练它：

```
# 加载库
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers

# 设定随机种子
np.random.seed(0)

# 设定想要的特征数量
number_of_features = 1000

# 从影评数据中加载数据和目标向量
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# 将影评数据转化为 one-hot 编码过的特征矩阵
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# 创建神经网络对象
network = models.Sequential()
```

```
# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu", input_shape=(
    number_of_features,)))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu"))

# 添加使用 sigmoid 激活函数的全连接层
network.add(layers.Dense(units=1, activation="sigmoid"))

# 编译神经网络
network.compile(loss="binary_crossentropy", # 交叉熵
                optimizer="rmsprop", # 均方根传播
                metrics=["accuracy"]) # 将准确率作为性能指标

# 训练神经网络
history = network.fit(features_train, # 特征
                      target_train, # 目标向量
                      epochs=3, # epoch 的数量
                      verbose=1, # 每个 epoch 之后打印描述
                      batch_size=100, # 每个批次中观察值的数量
                      validation_data=(features_test, target_test)) # 测试数据
```

```
Using TensorFlow backend.


Train on 25000 samples, validate on 25000 samples
Epoch 1/3
25000/25000 [==============================]
- 1s - loss: 0.4215 - acc: 0.8105 - val_loss: 0.3386 - val_acc: 0.8555
Epoch 2/
25000/25000 [==============================]
- 1s - loss: 0.3242 - acc: 0.8645 - val_loss: 0.3258 - val_acc: 0.8633
Epoch 3/3
25000/25000 [==============================]
- 1s - loss: 0.3121 - acc: 0.8699 - val_loss: 0.3265 - val_acc: 0.8599
```

讨论

在 20.2 节中，我们讨论过如何使用 Keras 的 sequential 模型，本节我们使用真实数据来训练神经网络。准确地说，我们用了 50,000 条影评数据（25,000 条用作训练数据，25,000 条用作测试数据），然后把它们分为正面评价和负面评价。我们把评论的文本转化为 1000 个二元特征值来表示 1000 个最常出现的词出现与否。简单地说，就是我们的神经网络会使用 25,000 个观察值来预测一条影评是正面的还是负面的评价，其中每个观

察值有 1000 个特征。

这里我们使用跟 20.2 节一样的神经网络（如果想看详细的解释，可以参考 20.2 节）。不一样的地方是，在 20.2 节中我们仅创建了神经网络但并没有训练它。

在 Keras 中，训练神经网络用的是 `fit` 方法，它需要指定 6 个重要的参数。前两个参数是训练数据的特征和目标向量。我们可以使用 `shape` 方法查看特征矩阵的形状：

```
# 查看特征矩阵的形状
features_train.shape

(25000, 1000)
```

`epochs` 参数指定在训练数据时需要使用多少个 epoch。`verbose` 参数决定在训练过程中需要输出多少信息：`0` 表示没有输出，`1` 表示输出一个进度条，`2` 表示每个 epoch 输出一条日志。`batch_size` 设定在计算多少个观察值之后才更新参数。

最后，我们保留一个测试集数据来评估模型。这些测试用的特征和目标向量可以放在 `validation_data` 参数中，用来评估模型。或者，我们可以使用 `validation_split` 参数来设定要留多少比例的训练集数据用于评估模型。

在 scikit-learn 中 `fit` 方法返回一个训练后的模型，但是在 Keras 中 `fit` 方法返回一个 `History` 对象，它包含损失数据和每个 epoch 的性能数据。

20.4 训练一个多元分类器

问题描述

训练一个多元分类器神经网络。

解决方案

使用 Keras 构建一个前馈神经网络，其输出层采用的是 softmax 激活函数：

```
# 加载库
import numpy as np
from keras.datasets import reuters
from keras.utils.np_utils import to_categorical
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
```

```
# 设定随机种子
np.random.seed(0)

# 设定我们想要的特征的数量
number_of_features = 5000

# 加载特征和目标数据
data = reuters.load_data(num_words=number_of_features)
(data_train, target_vector_train), (data_test, target_vector_test) = data

# 把特征数据转换成 one-hot 编码的特征矩阵
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# 把 one-hot 编码的特征向量转换成特征矩阵
target_train = to_categorical(target_vector_train)
target_test = to_categorical(target_vector_test)

# 启动神经网络
network = models.Sequential()

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=100,
                         activation="relu",
                         input_shape=(number_of_features,)))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=100, activation="relu"))

# 添加使用 softmax 激活函数的全连接层
network.add(layers.Dense(units=46, activation="softmax"))

# 编译神经网络
network.compile(loss="categorical_crossentropy", # 交叉熵
                optimizer="rmsprop", # 均方根传播
                metrics=["accuracy"]) # 将准确率作为性能指标

# 训练神经网络
history = network.fit(features_train, # 特征
                      target_train, # 目标向量
                      epochs=3, # 3 个 epoch
                      verbose=0, # 没有输出
                      batch_size=100, # 每个批次的观察值数量
                      validation_data=(features_test, target_test)) # 测试数据

Using TensorFlow backend.
```

讨论

在这次的解决方案中，我们创建了一个跟 20.3 节所介绍的二元分类器类似的神经网络，但还有一些不一样的地方。第一，这次的数据是 11,228 个路透社新闻专栏。每个新闻专栏被分成 46 个主题（即 46 个分类）。我们把新闻专栏转换为 5000 个二元特征值（分别表示在该专栏中这 5000 个词中的某个词是否出现过）来作为特征数据。我们用 one-hot 编码将目标数据转换为一个目标矩阵，每一行表示一个观察值属于 46 个分类中的哪一个：

```
# 查看目标矩阵
target_train


array([[ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.],
       ...,
       [ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.]])
```

第二，我们增加了隐藏层的神经元数量，帮助神经网络表示 46 个分类之间更复杂的关系。

第三，因为这是一个多元分类的问题，我们使用了有 46 个神经元（每个神经元对应一个分类）的输出层，其中包含一个 softmax 激活函数。这个激活函数会返回一个有 46 个值的矩阵，这 46 个值之和为 1。它们表示一个观察值被归类成 46 个分类之一的概率。

第四，我们使用了一个适合多元分类问题的损失函数，即分类交叉熵损失函数 `categorical_crossentropy`。

20.5 训练一个回归模型

问题描述

为回归问题训练一个神经网络。

解决方案

使用 Keras 构建只有一个输出神经元、没有激活函数的前馈神经网络：

```
# 加载库
import numpy as np
from keras.preprocessing.text import Tokenizer
from keras import models
```

```
from keras import layers
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn import preprocessing

# 设置随机种子
np.random.seed(0)

# 生成特征矩阵和目标向量
features, target = make_regression(n_samples = 10000,
                                   n_features = 3,
                                   n_informative = 3,
                                   n_targets = 1,
                                   noise = 0.0,
                                   random_state = 0)

# 把数据分为训练集和测试集
features_train, features_test, target_train, target_test = train_test_split(
features, target, test_size=0.33, random_state=0)

# 启动神经网络
network = models.Sequential()

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=32,
                         activation="relu",
                         input_shape=(features_train.shape[1],)))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=32, activation="relu"))

# 添加没有激活函数的全连接层
network.add(layers.Dense(units=1))

# 编译神经网络
network.compile(loss="mse", # 均方误差
                optimizer="RMSprop", # 优化算法
                metrics=["mse"]) # 均方误差

# 训练神经网络
history = network.fit(features_train, # 特征
                      target_train, # 目标向量
                      epochs=10, # epoch 的数量
                      verbose=0, # 没有输出
                      batch_size=100, # 每个批次的观察值数量
```

```
                    validation_data=(features_test, target_test)) # 测试数据
```

```
Using TensorFlow backend.
```

讨论

用神经网络预测连续的数值（而不是仅预测分类概率）是完全有可能做到的。在二元分类器中（见 20.3 节），我们使用只有一个神经元和一个 sigmoid 激活函数的输出层来预测观察值属于分类 1 的概率。重要的是，sigmoid 激活函数把输出值限定在 0 和 1 之间。如果我们不要激活函数，把这个限制去掉，就能输出连续的数值了。

另外，因为我们训练的是回归模型，所以应该使用合适的损失函数和性能评估指标。在我们的例子中使用的是均方误差（MSE）：

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

这里 n 是观察值的数量，y_i 是我们想预测的目标（观察值 i）的真实值 y；而 $\hat{y}_i$ 是模型预测的 y_i 值。

最后，因为我们是通过 scikit-learn 的 `make_regression` 方法产生的模拟数据，所以并不需要标准化特征。但是需要注意，在真实世界中，大多数情况下都是需要对数据做标准化处理的。

20.6 做预测

问题描述

使用神经网络做预测。

解决方案

使用 Keras 构建一个前馈神经网络，接着使用 `predict` 方法来做预测：

```
# 加载库
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
```

```
# 设置随机种子
np.random.seed(0)

# 设置想要的特征数量
number_of_features = 10000

# 加载 IMDB 电影的数据和目标向量
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# 把 IMDB 数据转换为 one-hot 编码的特征矩阵
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# 启动神经网络
network = models.Sequential()

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16,
                         activation="relu",
                         input_shape=(number_of_features,)))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu"))

# 添加使用 sigmoid 激活函数的全连接层
network.add(layers.Dense(units=1, activation="sigmoid"))

# 编译神经网络
network.compile(loss="binary_crossentropy", # 交叉熵
                optimizer="rmsprop", # 均方根传播
                metrics=["accuracy"]) # 将准确率作为性能指标

# 训练神经网络
history = network.fit(features_train, # 特征值
                      target_train, # 目标向量
                      epochs=3, # epoch 的数量
                      verbose=0, # 没有输出
                      batch_size=100, # 每个批次的观察值数量
                      validation_data=(features_test, target_test)) # 测试数据

# 预测测试集的分类
predicted_target = network.predict(features_test)
```

```
Using TensorFlow backend.
```

讨论

通过 Keras 做预测是一件很容易的事。一旦训练完神经网络，接着就可以使用 `predict` 方法了，它把一组特征当作输入参数，返回对每个观察值的预测值。在解决方案中，我们的神经网络被设定为做二元分类的预测，所以输出是预测观察值为分类 1 的概率。如果对观察值预测的值非常接近 1，则说明它很可能属于分类 1；反之，如果预测的值非常接近 0，那么观察值很可能属于分类 0。下面的代码查看的是测试集中第一个观察值属于分类 1 的预测概率：

```
# 查看第一个观察值属于分类 1 的预测概率
predicted_target[0]
```

```
array([ 0.83937484], dtype=float32)
```

20.7 可视化训练历史

问题描述

找到神经网络的损失（loss）或准确率分数的“甜蜜点（sweet spot）”。

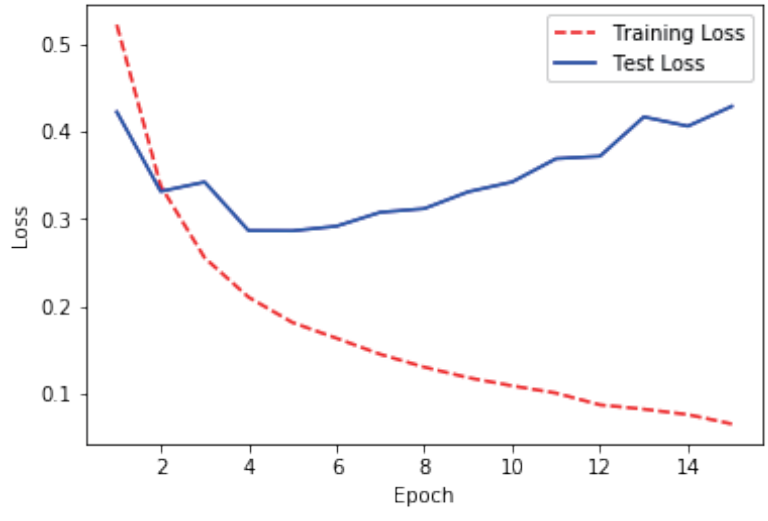
解决方案

使用 Matplotlib 可视化测试集和训练集在每个 epoch 上的损失（见代码后的图）：

```
# 加载库
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
import matplotlib.pyplot as plt

# 设置随机种子
np.random.seed(0)

# 设置我们想要的特征数量
number_of_features = 10000

# 从影评数据中加载数据和目标向量
```

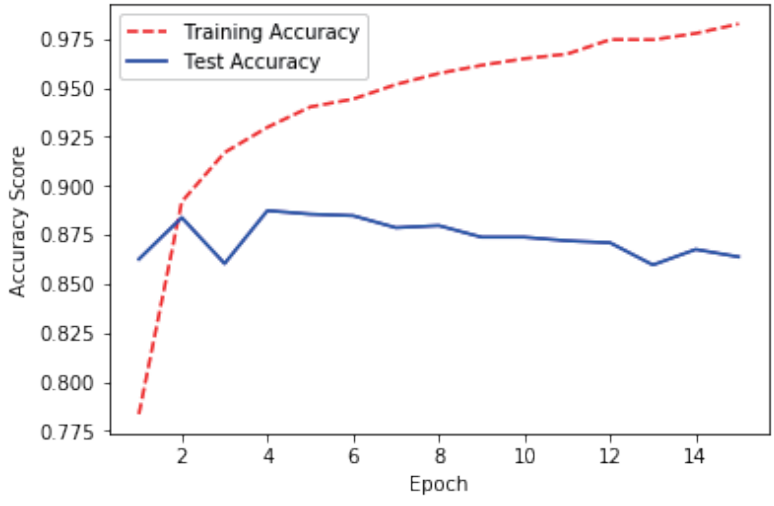
```
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# 把影评数据转换为 one-hot 编码的特征矩阵
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# 启动神经网络
network = models.Sequential()

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16,
                         activation="relu",
                         input_shape=(number_of_features,)))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu"))

# 添加使用 sigmoid 激活函数的全连接层
network.add(layers.Dense(units=1, activation="sigmoid"))

# 编译神经网络
network.compile(loss="binary_crossentropy", # 交叉熵
                optimizer="rmsprop", # 均方根传播
                metrics=["accuracy"]) # 将准确率作为性能指标

# 训练神经网络
history = network.fit(features_train, # 特征值
                      target_train, # 目标
                      epochs=15, # epoch 的数量
                      verbose=0, # 没有输出
                      batch_size=1000, # 每个批次的观察值数量
                      validation_data=(features_test, target_test)) # 测试数据

# 获取训练集和测试集的损失历史数值
training_loss = history.history["loss"]
test_loss = history.history["val_loss"]

# 为每个 epoch 创建编号
epoch_count = range(1, len(training_loss) + 1)

# 画出损失的历史数值
plt.plot(epoch_count, training_loss, "r--")
plt.plot(epoch_count, test_loss, "b-")
```

```
plt.legend(["Training Loss", "Test Loss"])
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show();

Using TensorFlow backend.
```



我们还可以使用同样的方法来可视化每个 epoch 的训练集和测试集数据的准确率（见代码后的图）：

```
# 获取训练集和测试集数据的准确率历史数值
training_accuracy = history.history["acc"]
test_accuracy = history.history["val_acc"]
plt.plot(epoch_count, training_accuracy, "r--")
plt.plot(epoch_count, test_accuracy, "b-")

# 可视化准确率的历史数值
plt.legend(["Training Accuracy", "Test Accuracy"])
plt.xlabel("Epoch")
plt.ylabel("Accuracy Score")
plt.show();
```



讨论

一开始我们的神经网络是新的，性能会比较差。而随着神经网络从训练数据中学习，其在训练集和测试集上的误差都会逐渐降低。但是，在某个时间点之后，神经网络开始“记住”训练数据，并且产生过拟合。当出现这种情况时，训练集误差会减小而测试集误差会增大。因此，在很多情形下会有一个“甜蜜点”，到达这个点时，测试集误差（这是我们主要关注的误差）最小。我们的解决方案将每个 epoch 的训练集和测试集损失可视化，因此可以很直观地看到这种现象。你可以看到，测试集误差在第 5 个 epoch 附近时是最小的，在这之后训练集损失继续减小而测试集损失开始增大。从这个点开始，模型就将出现过拟合。

20.8 通过权重调节减少过拟合

问题描述

减少过拟合。

解决方案

惩罚网络参数的过程，也被称为权重调节（*weight regularization*）：

```
# 加载库
import numpy as np
from keras.datasets import imdb
```

```
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
from keras import regularizers

# 设置随机种子
np.random.seed(0)

# 设置我们想要的特征数量
number_of_features = 1000

# 从影评数据中加载数据和目标向量
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# 把影评数据转换为 one-hot 编码的特征矩阵
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# 启动神经网络
network = models.Sequential()

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16,
                         activation="relu",
                         kernel_regularizer=regularizers.l2(0.01),
                         input_shape=(number_of_features,)))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16,
                         kernel_regularizer=regularizers.l2(0.01),
                         activation="relu"))

# 添加使用 sigmoid 激活函数的全连接层
network.add(layers.Dense(units=1, activation="sigmoid"))

# 编译神经网络
network.compile(loss="binary_crossentropy", # 交叉熵
                optimizer="rmsprop", # 均方根传播
                metrics=["accuracy"]) # 将准确率作为性能指标

# 训练神经网络
history = network.fit(features_train, # 特征
                      target_train, # 目标向量
```

```
                    epochs=3, # epoch 的数量
                    verbose=0, # 没有输出
                    batch_size=100, # 每个批次的观察值数量
                    validation_data=(features_test, target_test)) # 测试数据
```

```
Using TensorFlow backend.
```

讨论

对抗过拟合神经网络的策略之一就是惩罚神经网络的参数（即权重），使它们变成很小的值，从而创建一个更简单、更难过拟合的模型。这个方法叫作权重调节或者权重减少（weight decay）。更准确地说，权重调节就是将一个惩罚项加在L2范数这样的损失函数上。

在Keras中，我们可以通过在神经元层的参数中添加`kernel_regularizer=regularizers.l2(0.01)`来进行权重调节。其中，`0.01`表示要对参数值施加多重惩罚。

20.9 通过提前结束减少过拟合

问题描述

减少过拟合。

解决方案

当测试集损失不再减少时就结束训练，这个策略被称为提前结束（*early stopping*）：

```
# 加载库
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
from keras.callbacks import EarlyStopping, ModelCheckpoint

# 设置随机种子
np.random.seed(0)

# 设置我们想要的特征数量
number_of_features = 1000

# 从影评数据中加载数据和目标向量
```

```
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# 把影评数据转换为 one-hot 编码的特征矩阵
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# 启动神经网络
network = models.Sequential()

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16,
                         activation="relu",
                         input_shape=(number_of_features,)))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu"))

# 添加使用 sigmoid 激活函数的全连接层
network.add(layers.Dense(units=1, activation="sigmoid"))

# 编译神经网络
network.compile(loss="binary_crossentropy", # 交叉熵
                optimizer="rmsprop", # 均方根传播
                metrics=["accuracy"]) # 将准确率作为性能指标

# 设置一个回调函数来提前结束训练，并保存训练结束时的最佳模型
callbacks = [EarlyStopping(monitor="val_loss", patience=2),
             ModelCheckpoint(filepath="best_model.h5",
                             monitor="val_loss",
                             save_best_only=True)]

# 训练神经网络
history = network.fit(features_train, # 特征
                      target_train, # 目标向量
                      epochs=20, # epoch 的数量
                      callbacks=callbacks, # 提前结束
                      verbose=1, # 在每个 epoch 之后打印描述
                      batch_size=100, # 每个批次的观察值数量
                      validation_data=(features_test, target_test)) # 测试数据

Using TensorFlow backend.
```

讨论

正如我们在 20.7 节中讨论的，一般来说在前几个 epoch 中，训练集和测试集的误差都会减小，但是到了某个节点，神经网络会开始“记住”测试集数据，导致训练集误差持续减小而测试集误差开始增大。因为存在这种现象，所以一个最普遍且有效的对抗过拟合的方法就是，监视训练过程并且在测试集误差开始增大时就结束训练。这个策略被称作提前结束。

在 Keras 中，我们可以用一个回调函数来实施提前结束策略。回调函数是可以在训练过程的某几个特定阶段应用的函数，比如在每个 epoch 结束的时候。在本节的解决方案中，我们加入了 `EarlyStopping(monitor='val_loss', patience=2)`，告诉程序我们想监视每个 epoch 的测试集（验证集）损失，并且如果连续两个 epoch 测试集损失的情况都没有得到改善，就中断训练。但是因为我们设定了 `patience=2`，所以得不到最佳模型，而会得到最佳模型之后的两个 epoch 的模型。因此，我们也可以添加一个 `ModelCheckpoint` 操作，在每个检查点（如果你有一个持续多日的训练由于某种原因被打断，这时检查点就会很有用）之后把模型保存到文件中。这对我们会很有帮助，因为如果设定 `save_best_only=True`，`ModelCheckpoint` 就会仅保存最佳模型。

20.10 通过 Dropout 减少过拟合

问题描述

减少过拟合。

解决方案

使用 Dropout（丢弃）方法向网络结构中引入噪声：

```
# 加载库
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers

# 设置随机种子
np.random.seed(0)

# 设置我们想要的特征数量
number_of_features = 1000
```

```
# 从影评数据中加载数据和目标向量
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# 把影评数据转换为 one-hot 编码的特征矩阵
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# 启动神经网络
network = models.Sequential()

# 为输入层添加一个 Dropout 层
network.add(layers.Dropout(0.2, input_shape=(number_of_features,)))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu"))

# 为前面的隐藏层添加一个 Dropout 层
network.add(layers.Dropout(0.5))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu"))

# 为前面的隐藏层添加一个 Dropout 层
network.add(layers.Dropout(0.5))

# 添加使用 sigmoid 激活函数的全连接层
network.add(layers.Dense(units=1, activation="sigmoid"))

# 编译神经网络
network.compile(loss="binary_crossentropy", # 交叉熵
                optimizer="rmsprop", # 均方根传播
                metrics=["accuracy"]) # 将准确率作为性能指标

# 训练神经网络
history = network.fit(features_train, # 特征
                      target_train, # 目标向量
                      epochs=3, # epoch 的数量
                      verbose=0, # 没有输出
                      batch_size=100, # 每个批次的观察值数量
                      validation_data=(features_test, target_test)) # 测试数据
```

```
Using TensorFlow backend.
```

讨论

Dropout 是一个流行的、强大的调节神经网络的方法。在 Dropout 方法中，每创建一个批次的观察值用于训练时，一层或者多层的一部分神经元就会被乘以 0（即被丢弃）。虽然每个批次都是在同一个网络中训练的（比如，有同样的参数），但是每个批次面对的网络结构都有些许差异。

Dropout 方法很有效是因为它不断随机地丢弃每个批次中的神经元，迫使神经元在各种网络结构下依然能够学习参数。换句话说，神经元们变得对其他隐藏的神经元的中断（也可以理解为噪声）更加健壮，这样可以阻止网络记住训练集数据。

在隐藏层和输入层都可以添加 Dropout 方法。当一个输入层被丢弃后，它的特征值就不会在那个批次中被传进网络。一般对神经元丢弃的比例为，输入层 0.2，隐藏层 0.5。

在 Keras 中，可以通过在网络架构中添加若干个 `Dropout` 层来实现 Dropout 方法。每个 `Dropout` 层会在每个批次中丢弃前一层传过来的用户定义数量的神经元，这个数量是超参数。记住，在 Keras 中输入层被认定为第一层，并且不需要用 `add` 方法来添加。所以，如果想给输入层添加 Dropout 方法，给网络结构加的第一层就是 `Dropout` 层。这一层既包含输入层神经元的丢弃比例 `0.2`，也包含定义观察值的形状 `input_shape`。接着，我们在每个隐藏层之后添加一个神经元丢弃比例为 `0.5` 的 `Dropout` 层。

20.11 保存模型训练过程

问题描述

假设有一个神经网络需要花很长时间来训练，现在需要保存训练过程以防训练被中断。

解决方案

在每个 epoch 之后使用回调函数 `ModelCheckpoint` 来保存模型：

```
# 加载库
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
from keras.callbacks import ModelCheckpoint

# 设定随机种子
np.random.seed(0)
```

```
# 设置我们想要的特征数量
number_of_features = 1000

# 从影评数据中加载数据和目标向量
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# 把影评数据转换为 one-hot 编码的特征矩阵
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# 启动神经网络
network = models.Sequential()

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16,
                         activation="relu",
                         input_shape=(number_of_features,)))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu"))

# 添加使用 sigmoid 激活函数的全连接层
network.add(layers.Dense(units=1, activation="sigmoid"))

# 编译神经网络
network.compile(loss="binary_crossentropy", # 交叉熵
                optimizer="rmsprop", # 均方根传播
                metrics=["accuracy"]) # 将准确率作为性能指标

# 设置一个回调函数来提前结束训练，并保存训练结束时的最佳模型
checkpoint = [ModelCheckpoint(filepath="models.hdf5")]

# 训练神经网络
history = network.fit(features_train, # 特征
                      target_train, # 目标向量
                      epochs=3, # epoch 的数量
                      callbacks=checkpoint, # 检查点
                      verbose=0, # 没有输出
                      batch_size=100, # 每个批次的观察值数量
                      validation_data=(features_test, target_test)) # 测试数据

Using TensorFlow backend.
```

讨论

在 20.8 节中，当测试集的误差不再继续减小时，我们使用回调函数 `ModelCheckpoint` 联合 `EarlyStopping` 来结束监控和训练。但是，使用 `ModelCheckpoint` 还有一个更实际的原因。在真实世界中，神经网络一般需要训练几小时甚至几天。在这段时间内，有很多环节都可能出错：电脑可能没电，服务器可能崩溃，或者某个冒失鬼关闭了你的笔记本电脑。

`ModelCheckpoint` 会在每一个 epoch 之后保存模型，以避免这一类问题。具体来说，就是在每个 epoch 之后，`ModelCheckpoint` 把模型保存到 `filepath` 参数指定的路径中。如果只给定一个文件名（比如解决方案中的 *models.hdf5*），那么这个文件在每个 epoch 后都会被最近的模型重写。如果只想根据某个损失函数的表现来保存最佳模型，可以设置 `save_best_only=True` 和 `monitor='val_loss'`，这样如果现有模型比前一个模型的测试集损失更大的话，该文件并不会被重写。还有一种方案，我们可以保存每个 epoch 的模型，单独作为一个文件，并且将 epoch 编号和测试集损失值写在文件名中。比方说，设置 `filepath` 参数为 `model_{epoch:02d}_{val_loss:.2f}.hdf5`，如果在第 11 个 epoch 之后保存，测试集损失值是 0.33 的话，包含这个模型的文件的名字就是 *model_10_0.33.hdf5*（注意，这里 epoch 的编号是从 0 开始的）。

20.12 使用 *k* 折交叉验证评估神经网络

问题描述

使用 *k* 折交叉验证评估一个神经网络。

解决方案

一般来说，没有必要使用 *k* 折交叉验证评估神经网络，也不推荐使用。但如果是以下的情况，那么可以使用 *k* 折交叉验证评估神经网络：用 Keras 的 scikit-learn 封装器让 Keras 的 sequential 模型使用 scikit-learn 的 API。

```
# 加载库
import numpy as np
from keras import models
from keras import layers
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.datasets import make_classification
```

```
# 设定随机种子
np.random.seed(0)

# 特征数量
number_of_features = 100

# 生成特征矩阵和目标向量
features, target = make_classification(n_samples = 10000,
                                       n_features = number_of_features,
                                       n_informative = 3,
                                       n_redundant = 0,
                                       n_classes = 2,
                                       weights = [.5, .5],
                                       random_state = 0)

# 创建一个函数，返回编译过的网络
def create_network():

    # 启动神经网络
    network = models.Sequential()

    # 添加使用 ReLU 激活函数的全连接层
    network.add(layers.Dense(units=16, activation="relu", input_shape=(
        number_of_features,)))

    # 添加使用 ReLU 激活函数的全连接层
    network.add(layers.Dense(units=16, activation="relu"))

    # 添加使用 sigmoid 激活函数的全连接层
    network.add(layers.Dense(units=1, activation="sigmoid"))

    # 编译神经网络
    network.compile(loss="binary_crossentropy", # 交叉熵
                    optimizer="rmsprop", # 均方根传播
                    metrics=["accuracy"]) # 将准确率作为性能指标

    # 返回编译过的网络
    return network

# 封装 Keras 模型，以便它能被 scikit-learn 使用
neural_network = KerasClassifier(build_fn=create_network,
                                 epochs=10,
                                 batch_size=100,
                                 verbose=0)
```

```
# 使用 3 折交叉验证来评估神经网络
cross_val_score(neural_network, features, target, cv=3)

Using TensorFlow backend.

array([ 0.90461907, 0.77437743, 0.87068707])
```

讨论

从理论上来讲，没有任何理由不能使用交叉验证评估神经网络。但是，神经网络经常用于非常大的数据集而且可能需要几小时甚至几天来训练，所以如果训练时间很长的话，采用 *k* 折交叉验证就会增加计算开销，这并不是一个值得推荐的做法。比如，对于一个通常需要一天来训练的模型，如果使用 10 折交叉验证评估，就要花 10 天的时间。如果数据量非常大，常见的稳妥做法是用某个测试集评估神经网络。

如果数据规模不大，*k* 折交叉验证可以被用于最大化我们评估神经网络性能的能力。在 Keras 中，这是可行的，因为我们可以“封装”所有的神经网络，使它们能使用 scikit-learn 中的评估特性，包括 *k* 折交叉验证。为了实现这一点，必须先创建一个函数，返回编译好的神经网络。接着，使用 `KerasClassifier`（这里假设有一个分类器，如果是回归分类器，就应该使用 `KerasRegressor`）来封装模型，使它可以被 scikit-learn 使用。然后，神经网络就可以像其他 scikit-learn 的学习算法（比如随机森林、逻辑回归）一样被使用了。在本节的解决方案中，我们对神经网络使用了 `cross_val_score` 来运行一个 3 折交叉验证。

20.13 调校神经网络

问题描述

为神经网络自动选择最优超参数。

解决方案

联合使用 Keras 神经网络和 scikit-learn 的模型选择工具（比如 `Grid SearchCV`）：

```
# 加载库
import numpy as np
from keras import models
from keras import layers
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import make_classification
```

```
# 设置随机种子
np.random.seed(0)

# 特征的数量
number_of_features = 100

# 生成特征矩阵和目标向量
features, target = make_classification(n_samples = 10000,
                                       n_features = number_of_features,
                                       n_informative = 3,
                                       n_redundant = 0,
                                       n_classes = 2,
                                       weights = [.5, .5],
                                       random_state = 0)

# 创建一个函数，返回编译过的神经网络
def create_network(optimizer="rmsprop"):

    # 启动神经网络
    network = models.Sequential()

    # 添加使用 ReLU 激活函数的全连接层
    network.add(layers.Dense(units=16,
                             activation="relu",
                             input_shape=(number_of_features,)))

    # 添加使用 ReLU 激活函数的全连接层
    network.add(layers.Dense(units=16, activation="relu"))

    # 添加使用 sigmoid 激活函数的全连接层
    network.add(layers.Dense(units=1, activation="sigmoid"))

    # 编译神经网络
    network.compile(loss="binary_crossentropy", # 交叉熵
                    optimizer=optimizer, # 优化器
                    metrics=["accuracy"]) # 将准确率作为性能指标

    # 返回编译过的网络
        return network

# 封装 Keras 模型以便它能被 scikit-learn 使用
neural_network = KerasClassifier(build_fn=create_network, verbose=0)

# 创建超参数空间
epochs = [5, 10]
```

```
batches = [5, 10, 100]
optimizers = ["rmsprop", "adam"]

# 创建超参数选项
hyperparameters = dict(optimizer=optimizers, epochs=epochs, batch_
size=batches)

# 创建网格搜索
grid = GridSearchCV(estimator=neural_network, param_grid=hyperparameters)

# 实现网格搜索
grid_result = grid.fit(features, target)

Using TensorFlow backend.
```

讨论

第 12 章的 12.1 和 12.2 节讨论了如何使用 scikit-learn 的模型选择技术为 scikit-learn 模型寻找最优超参数。在 20.12 节中我们学习了封装神经网络，以使其可以使用 scikit-learn 的 API。本节我们联合使用这两个技术以找出神经网络的最优超参数。

对于一个模型而言，超参数是非常重要的，应该仔细地选择。但是，在你准备采用网格搜索这样的模型选择策略之前，必须认识到，如果你的模型需要 12 个小时或者一天的时间来训练，那么网格搜索可能需要花上一周甚至更长的时间。因此，神经网络自动超参数调校不是万能药，但是它在某些特定情形下是有用的。

在上面的解决方案中，我们对一些超参数选项（包括优化算法、epoch 的数量和批次的大小）实施了交叉验证的网格搜索。即使这个玩具例子（toy example）都花了几分钟来运行，但是一旦运行结束，我们就可以使用 `best_params_` 方法来查看产生了最优神经网络的超参数：

```
# 查看最优神经网络的超参数
grid_result.best_params_

{'batch_size': 10, 'epochs': 5, 'optimizer': 'adam'}
```

20.14 可视化神经网络

问题描述

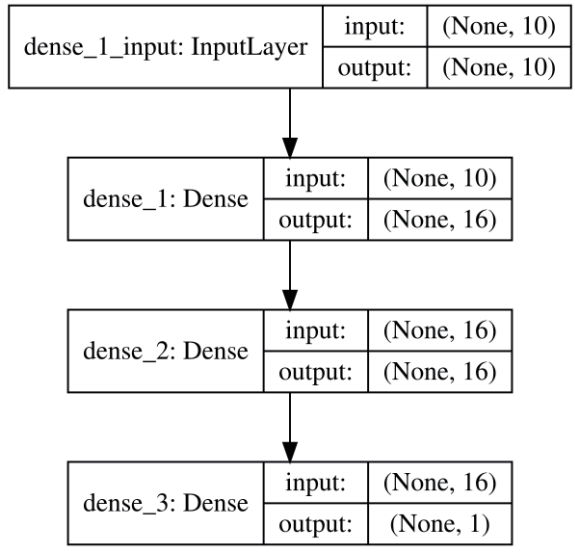
快速可视化神经网络的结构。

解决方案

使用 Keras 的 `model_to_dot` 或者 `plot_model`（见代码后的图）：

```
# 加载库
from keras import models
from keras import layers
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
from keras.utils import plot_model

# 启动神经网络
network = models.Sequential()

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu", input_shape=(10,)))

# 添加使用 ReLU 激活函数的全连接层
network.add(layers.Dense(units=16, activation="relu"))

# 添加使用 sigmoid 激活函数的全连接层
network.add(layers.Dense(units=1, activation="sigmoid"))

# 可视化网络结构
SVG(model_to_dot(network, show_shapes=True).create(prog="dot", format="svg"))
```

```
Using TensorFlow backend.
```



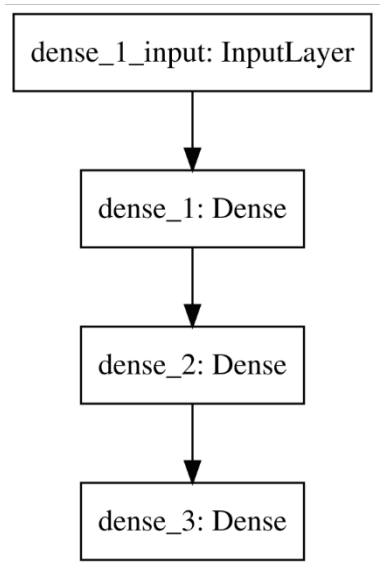
如果想把上面的网络结构图保存为文件，可以使用 `plot_model`：

```
# 将可视化后的网络结构图保存为文件
plot_model(network, show_shapes=True, to_file="network.png")
```

讨论

Keras 提供了工具函数用于快速可视化神经网络。如果想在 Jupyter Notebook 中显示一个神经网络，可以使用 `model_to_dot`。`show_shapes` 参数指定是否展示输入和输出的形状，它可以帮助我们调试网络。如果想展示一个更简单的模型，可以设置 `show_shapes=False`（见代码后的图）：

```
# 画出网络结构
SVG(model_to_dot(network, show_shapes=False).create(prog="dot",
format="svg"))
```



20.15 图像分类

问题描述

使用卷积神经网络对一组图像分类。

解决方案

使用 Keras 创建至少有一个卷积层的神经网络：

```
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.utils import np_utils
from keras import backend as K

# 设置色彩通道值优先
K.set_image_data_format("channels_first")

# 设置随机种子
np.random.seed(0)
```

```
# 图像信息
channels = 1
height = 28
width = 28

# 从 MNIST 数据集中读取数据和目标
(data_train, target_train), (data_test, target_test) = mnist.load_data()

# 将训练集图像数据转换成特征
data_train = data_train.reshape(data_train.shape[0], channels, height, width)

# 将测试集图像数据转换成特征
data_test = data_test.reshape(data_test.shape[0], channels, height, width)

# 将像素的强度值收缩到 0 和 1 之间
features_train = data_train / 255
features_test = data_test / 255

# 对目标进行 one-hot 编码
target_train = np_utils.to_categorical(target_train)
target_test = np_utils.to_categorical(target_test)
number_of_classes = target_test.shape[1]

# 启动神经网络
network = Sequential()

# 添加有 64 个过滤器、一个大小为 5x5 的窗口和 ReLU 激活函数的卷积层
network.add(Conv2D(filters=64,
                   kernel_size=(5, 5),
                   input_shape=(channels, width, height),
                   activation='relu'))

# 添加带一个 2x2 窗口的最大池化层
network.add(MaxPooling2D(pool_size=(2, 2)))

# 添加 Dropout 层
network.add(Dropout(0.5))

# 添加一层来压平输入
network.add(Flatten())

# 添加带 ReLU 激活函数的有 128 个神经元的全连接层
network.add(Dense(128, activation="relu"))

# 添加 Dropout 层
```

```
network.add(Dropout(0.5))

# 添加使用 softmax 激活函数的全连接层
network.add(Dense(number_of_classes, activation="softmax"))

# 编译神经网络
network.compile(loss="categorical_crossentropy", # 交叉熵
                optimizer="rmsprop", # 均方根传播
                metrics=["accuracy"]) # 将准确率作为性能指标

# 训练神经网络
network.fit(features_train, # 特征
            target_train, # 目标向量
            epochs=2, # epoch 的数量
            verbose=0, # 没有输出
            batch_size=1000, # 每个批次的观察值数量
            validation_data=(features_test, target_test)) # 测试数据

Using TensorFlow backend.

<keras.callbacks.History at 0x133f37e80>
```

讨论

卷积神经网络（Convolutional Neural Networks，又被简写为 ConvNets）是一种流行的神经网络，而且被证明在计算机视觉领域非常有效（比如能识别猫、狗、飞机甚至热狗）。对图像使用前馈神经网络是完全可行的，图像的每个像素都是一个特征。但是如果这么做，会遇到两个大问题。首先，前馈神经网络并没有考虑像素之间的空间结构。比如对于一张 10×10 像素的图像，我们可能会把它转换成一个有 100 个像素特征的向量，这样的话在前向传播时，就会认为第 1 个特征（比如像素值）和第 10 个特征之间的空间关系与它和第 11 个特征之间的空间关系是一样的。但事实上第 10 个特征代表的是处在图片远端的像素，距离第 1 个特征代表的像素较远，而第 11 个特征表示的是第 1 个像素正下方的像素。其次，前馈神经网络学习的是特征的全局关系而不是局部的模式。更具体地说，这意味着前馈神经网络无法识别一个物体，不管该物体出现在图像的何处。举例来说，假设我们要训练一个神经网络来识别脸，而这些脸可能出现在图像的任意地方，从右上方到中间再到左下方。

卷积神经网络的强大之处在于，它可以解决上面的两个问题（还可以解决其他问题）。本书不会全面解释卷积神经网络，这里仅做简单的说明。一张图像的数据通常包含两个或者三个维度：高度、宽度和深度。前两个维度非常明显，最后一个需要做一些解释。深度是指一个像素的颜色。在灰度图像中只有一个深度值（像素的强度），因此一张图像

可以表示为一个矩阵。但是在彩色图像中，一个像素的颜色是由多个值表示的。比如，在一张 RGB 图像中，一个像素的颜色是用 3 个值表示的，分别对应 3 个颜色分量（红、绿、蓝）。因此，一张图像的数据可以被想象成一个 3 维张量：宽度 × 高度 × 深度（又被叫作特征图）。在卷积神经网络中，卷积可以被想象成在图像的像素上滑动一个窗口，通过这个窗口查看像素和它周围的邻居们。接着，它把初始图像数据转换为一个新的 3 维张量，头两个张量是近似的宽度和高度，而第 3 个维度（包含颜色值）现在表示像素“属于”哪种模式（比如，尖角或者梯度渐变，它也被叫作过滤器）。

第二个重要的概念是池化层。池化层会在我们的数据上移动一个窗口（通常窗口是按每 *n* 个像素作为一个步长来移动的，叫作 *striding*），然后对窗口里的数据以某种方式求和，以缩减数据规模。最常用的方法是最大池化（max pooling），它把每个窗口的最大值传递到下一层。使用最大池化的原因之一是它很实用，卷积过程产生了很多要学习的参数，这会让学习过程很快就没有什么收获，所以通过最大池化减少参数的数量是有益的。可以直观地将最大池化想象成“缩小”图像。

这里举一个例子来说明可能会有助于理解。假设我们有一张包含一只狗的脸的图像。第一个卷积层可能找到一些模式，比如形状的边缘。接着，我们使用最大池化层来“缩小”图像，然后用第二个卷积层找到另一些模式，比如狗的耳朵。最后，我们使用第三个最大池化层进一步缩小图像，再使用一个卷积层来找到像狗的脸这样的模式。

全连接层经常被用在网络的最后来做分类。

尽管我们的解决方案看起来有很多行代码，但其实它跟本章前面讲的二元分类器非常类似。在这个解决方案中我们使用了著名的 MNIST 数据集，它实际上是机器学习领域的一个指标性数据集。MNIST 数据集包含 70,000 张手写的 0 ~ 9 的数字的小图像（28 × 28 像素）。数据集已经做过标记，所以我们知道每一张小图像上的真实数字（也就是其分类）。标准的训练集和测试集数据的配比是，用 60,000 张图像做训练，用 10,000 张图像做测试。

我们把数据重新组织成卷积神经网络期望的格式。具体来讲，就是使用 `reshape` 把样本数据转换成 Keras 期望的形状。接着，我们把数据的值调节到 0 和 1 之间，因为如果样本值比网络的参数（通常都初始化成较小的数值）大很多，训练后网络的性能会很差。最后，我们把目标数据用 one-hot 编码，这样样本的目标就有 10 个分类，代表从 0 到 9 的数字。

像这样处理图像数据之后，我们就可以创建卷积神经网络了。第一步，添加一个卷积层，并指定过滤器的数量和其他特性。窗口的大小是一个超参数，不过 3 × 3 的窗口对于大部分的图像来说都适用。一般图像越大，使用的窗口就越大。第二步，添加最大池化层，

对相邻的像素求和。第三步，添加 Dropout 层来减小过拟合的概率。第四步，添加一个压平层把卷积输入转换成全连接层可用的格式。最后，添加全连接层和输出层对数据进行分类。注意，因为这是一个多元分类问题，所以我们在输出层使用的是 softmax 激活函数。

需要指出，这是一个很简单的卷积神经网络，在实际应用中常常能看到有更多卷积层和最大池化层的更深的网络。

20.16 通过图像增强来改善卷积神经网络的性能

问题描述

改善卷积神经网络的性能。

解决方案

为了得到更好的结果，对图像进行预处理，并使用 ImageDataGenerator 提前增强数据：

```
# 加载库
from keras.preprocessing.image import ImageDataGenerator

# 创建图像增强对象
augmentation = ImageDataGenerator(featurewise_center=True, # 实施 ZCA 白化
                                  zoom_range=0.3, # 随机放大图像
                                  width_shift_range=0.2, # 随机打乱图像
                                  horizontal_flip=True, # 随机翻转图像
                                  rotation_range=90) # 随机旋转图像

# 对 raw/images 文件夹下所有的图像进行处理
augment_images = augmentation.flow_from_directory("raw/images", # 图像文件夹
                                                  batch_size=32, # 批次的大小
                                                  class_mode="binary", # 分类
                                                  save_to_dir="processed/
images")

Using TensorFlow backend.

Found 12665 images belonging to 2 classes.
```

讨论

先给大家道个歉，因为这个解决方案的代码不能直接运行，原因是你们没有这个图像文

件夹。但是，你有一个图像文件夹，而且你想对里面的图像进行处理，这样的情形却是很常见的。因此，这个技术可以很容易地应用到你自己的图像上。

改善卷积神经网络性能的一个方法是预处理图像。我们在第 8 章中已经讨论了一些技巧，尽管如此，Keras 的 `ImageDataGenerator` 仍然值得强调一下，它包含一些基础的预处理技术。比如，我们的解决方案中使用了 `featurewise_center=True` 来标准化整个数据集中的像素。

第二个改善卷积神经网络性能的技术是加入噪声。神经网络有一个有趣的特征，就是在数据中加入噪声后，网络的性能反而会变好。这是因为添加的噪声可以让神经网络对真实世界的噪声变得更具鲁棒性，并且能防止神经网络过拟合。

当对图像使用卷积神经网络进行训练的时候，可以通过多种方法随机转换图像，以实现向样本数据中加入噪声，比如镜像翻转图像，或者局部放大图像。即使很小的变化，也可以显著改善模型的性能。我们可以使用同样的 `ImageDataGenerator` 类实施这些转换。Keras 文档（参见本节末尾的“延伸阅读”）列出了所有可用的转换方式。我们的解决方案就是一个很好的例子，其中包含图像的随机缩放、平移、翻转和旋转。

有一点很重要，`flow_from_directory` 的输出是一个 Python 的生成器对象。这是因为大多数情况下我们希望在需要时，即当图像被输入神经网络中进行训练的时候，才进行处理。如果想在训练之前先处理所有图像，可以简单地遍历生成器。

最后，因为 augment_images 是一个生成器，所以在训练神经网络时，必须使用 fit_generator 而不是 fit。举个例子：

```
# 训练神经网络
network.fit_generator(augment_images,
                      # 在每个 epoch 中调用生成器的次数
                      steps_per_epoch=2000,
                      # epoch 的数量
                      epochs=5,
                      # 测试数据生成器
                      validation_data=augment_images_test,
                      # 在每个测试 epoch 中调用生成器的次数
                      validation_steps=800)
```

本节所用的全部原始图片都可以在 GitHub（*http://bit.ly/2FvrVVq*）上下载。

延伸阅读

- Keras 文档：图像预处理（*https://keras.io/preprocessing/image/*）

20.17 文本分类

问题描述

对文本数据分类。

解决方案

使用长短期记忆递归神经网络：

```
# 加载库
import numpy as np
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras import models
from keras import layers

# 设置随机种子
np.random.seed(0)

# 设置想要的特征数量
number_of_features = 1000

# 从影评数据中加载数据和目标向量
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# 采用添加填充值或者截断的方式，使每个样本都有 400 个特征
features_train = sequence.pad_sequences(data_train, maxlen=400)
features_test = sequence.pad_sequences(data_test, maxlen=400)

# 启动神经网络
network = models.Sequential()

# 添加嵌入层
network.add(layers.Embedding(input_dim=number_of_features, output_dim=128))

# 添加一个有 128 个神经元的长短期记忆网络层
network.add(layers.LSTM(units=128))

# 添加使用 sigmoid 激活函数的全连接层
network.add(layers.Dense(units=1, activation="sigmoid"))
```

```
# 编译神经网络
network.compile(loss="binary_crossentropy", # 交叉熵
                optimizer="Adam", # Adam 优化器
                metrics=["accuracy"]) # 将准确率作为性能指标

# 训练神经网络
history = network.fit(features_train, # 特征
                      target_train, # 目标向量
                      epochs=3, # epoch 的数量
                      verbose=0, # 没有输出
                      batch_size=1000, # 每个批次的样本数量
                      validation_data=(features_test, target_test)) # 测试数据
```

```
Using TensorFlow backend.
```

讨论

我们经常会遇到文本数据分类的问题，尽管可以使用一种卷积神经网络来实现，但本节将聚焦于一个更流行的选择：递归神经网络。递归神经网络有一个关键特性，即信息在网络中回环，这让递归神经网络拥有一种记忆，使它可以更好地理解顺序数据。一种流行的递归神经网络是长短期记忆递归（Long Short-Term Memory，LSTM）神经网络。关于 LSTM 更详细的解释请参考其他相关资料。

在解决方案中，我们使用了 20.3 节的影评数据，想训练一个 LSTM 神经网络来预测这些影评是正面的还是负面的评价。在训练网络之前，需要对数据进行一些处理。我们的文本数据是一个整数列表：

```
# 查看第一个样本
print(data_train[0])
```

```
[1, 14, 22, 16, 43, 530, 973, 2, 2, 65, 458, 2, 66, 2, 4, 173, 36, 256, 5,
25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112,
167, 2, 336, 385, 39, 4, 172, 2, 2, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6,
147, 2, 19, 14, 22, 4, 2, 2, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76,
15, 13, 2, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316,
8, 106, 5, 4, 2, 2, 16, 480, 66, 2, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124,
51, 36, 135, 48, 25, 2, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82,
2, 8, 4, 107, 117, 2, 15, 256, 4, 2, 7, 2, 5, 723, 36, 71, 43, 530, 476, 26,
400, 317, 46, 7, 4, 2, 2, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2, 56, 26,
141, 6, 194, 2, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 2, 18,
51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 2, 88, 12, 16, 283, 5, 16,
2, 113, 103, 32, 15, 16, 2, 19, 178, 32]
```

列表中的每个整数对应一个词，但是因为每条影评包含的词的数量不一样，所以对应的

样本长度不一。因此，在将这些数据输入神经网络之前，需要让所有的样本具有同样的长度。我们使用 pad_sequences 来实现这一点，pad_sequences 为每个样本加入填充值使它们长度相同。可以查看第一个样本经过 pad_sequences 处理之后的效果：

```
# 查看第一个样本
print(features_test[0])
```

```
[ 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   1  89  27   2   2  17 199 132   5   2  16   2  24   8
760   4   2   7   4  22   2   2  16   2  17   2   7   2   2   9   4   2
  8  14 991  13 877  38  19  27 239  13 100 235  61 483   2   4   7   4
 20 131   2  72   8  14 251  27   2   7 308  16 735   2  17  29 144  28
 77   2  18  12]
```

接着，我们使用自然语言处理领域效果最好的技术之一：词向量。在 20.3 节的二元分类器中，我们对样本进行 one-hot 编码后作为神经网络的输入。但是这里我们把每个词表示为一个多维空间的向量，并且用两个向量之间的距离表示其对应的两个词的相似度。在 Keras 中，可以通过加入一个 Embedding 层来实现。对于每一个传入 Embedding 层的值，Keras 都会输出一个向量来表示这个词。紧接着的一层是有 128 个神经元的 LSTM 层，它使之前输入的信息能在未来使用，非常适合处理顺序数据。最后，因为这是一个二元分类问题（每一条影评要么是正面的要么是负面的），所以加入只有一个神经元和 sigmoid 激活函数的全连接输出层。

这里需要声明，LSTM 是一个很大的话题，也是很多研究的关注点。希望本节内容对各位读者有用，尽管离 LSTM 的真实应用还相距很远。

第 21 章

保存和加载训练后的模型

21.0 简介

前面的 20 章有将近 200 个小节，我们介绍了怎样获取原始数据，以及如何使用机器学习创建高性能的预测模型。但是，为了让这些工作有价值，我们需要用模型做一些事情，比如把它整合到一个已有的应用里。要实现这个目标，就必须在训练后保存模型，并且在应用需要的时候加载它。这就是本书最后一章的重点。

21.1 保存和加载 scikit-learn 模型

问题描述

有一个训练好的 scikit-learn 模型，你想保存它并且在其他地方加载它。

解决方案

把模型保存为 pickle 文件：

```
# 加载库
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets
from sklearn.externals import joblib

# 加载数据
iris = datasets.load_iris()
features = iris.data
target = iris.target
```

```
# 创建决策树分类器对象
classifer = RandomForestClassifier()

# 训练模型
model = classifer.fit(features, target)

# 把模型保存为 pickle 文件
joblib.dump(model,"model.pkl")

['model.pkl']
```

一旦模型被保存，我们就可以在目标应用（比如 Web 应用）上使用 scikit-learn 来加载模型：

```
# 从文件中加载模型
classifer = joblib.load("model.pkl")
```

然后使用它来做预测：

```
# 创建新的样本
new_observation = [[ 5.2, 3.2, 1.1, 0.1]]

# 预测样本的分类
classifer.predict(new_observation)

array([0])
```

讨论

在生产中使用模型的第一步，就是把模型保存成文件，使它可以被另一个应用或者工作流加载。把模型保存为 pickle 文件就能达到这个目的。pickle 是 Python 特有的数据格式。具体来讲，我们使用 joblib（它是一个库，作用是让 pickle 文件适用于 NumPy 数组很大的情况，在 scikit-learn 训练后的模型中，这种情况很常见）来保存模型。

当保存 scikit-learn 模型时，要留心，因为你所保存的模型有可能在各个版本的 scikit-learn 中不兼容，所以在文件名中写上模型所用的 scikit-learn 版本就会很有用：

```
# 加载库
import sklearn

# 获得 scikit-learn 的版本
scikit_version = joblib.__version__

# 把模型保存为 pickle 文件
joblib.dump(model,"model_{version}.pkl".format(version=scikit_version))

['model_0.11.pkl']
```

21.2 保存和加载 Keras 模型

问题描述

有一个训练好的 Keras 模型，你想保存它并在别处加载它。

解决方案

将模型保存为 HDF5 文件：

```
# 加载库
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
from keras.models import load_model

# 设置随机种子
np.random.seed(0)

# 设定想要的特征数量
number_of_features = 1000

# 从影评数据中加载数据和目标向量
(train_data, train_target), (test_data, test_target) = imdb.load_data(
    num_words=number_of_features)

# 把影评数据转换为 one-hot 编码的特征矩阵
tokenizer = Tokenizer(num_words=number_of_features)
train_features = tokenizer.sequences_to_matrix(train_data, mode="binary")
test_features = tokenizer.sequences_to_matrix(test_data, mode="binary")

# 启动神经网络
network = models.Sequential()

# 添加使用 ReLu 激活函数的全连接层
network.add(layers.Dense(units=16,
                         activation="relu",
                         input_shape=(number_of_features,)))

# 添加使用 sigmoid 激活函数的全连接层
network.add(layers.Dense(units=1, activation=" sigmoid" ))
```

```
# 编译神经网络
network.compile(loss="binary_crossentropy", # 交叉熵
                optimizer=optimizer, # 优化器
                metrics=["accuracy"]) # 将准确率作为性能指标

# 训练神经网络
history = network.fit(train_features, # 特征值
                      target_train, # 目标向量
                      epochs=3, # epoch 数量
                      verbose=0, # 没有输出
                      batch_size=100, # 每个批次的观察值数量
                      validation_data=(test_features, test_target)) # 测试数据

# 保存神经网络
network.save("model.h5")

Using TensorFlow backend.
```

然后，就可以在另一个应用或者其他训练中加载这个模型：

```
# 加载神经网络
network = load_model("model.h5")
```

讨论

不同于 scikit-learn，Keras 不推荐使用 pickle 格式来保存模型，而是要将模型保存为 HDF5 文件。HDF5 文件包含了你需要的一切，不仅包含加载模型做预测所需要的结构和训练后的参数，而且包含重新训练所需要的各种设置（即损失、优化器的设置和当前状态）。

关于作者

Chris Albon 是一位有十年经验的数据科学家和政治学家，他将统计学习、人工智能和软件工程应用到政治和社会活动以及人道主义活动中，譬如监查选举情况、灾难救助等。目前，Chris 是肯尼亚创业公司 BRCK 的首席数据科学家。这家公司致力于为前沿市场的互联网用户构建一个稳健的网络。

封面说明

本书封面上的动物是绿颊非洲咬鹃（Narina trogon 或 Apaloderma narina），Narina trogon 是以法国鸟类学家 Fran Çois Levaillant 情妇的名字命名的，她的名字取自科伊科伊（Khoikhoi）语中的一个词，意为“花朵”，不过这个词读起来很费劲。绿颊非洲咬鹃在非洲地区很常见，无论是低地还是高原，无论是热带还是温带气候，它们都能适应。这些鸟儿通常把巢穴筑在树洞里。它们分布广泛，因此也很少被认为需要保护。

绿颊非洲咬鹃主要以昆虫、小型无脊椎动物、小型啮齿动物和小型爬行动物为食。雄鸟羽毛鲜亮，会不断发出刺耳但低声的鸣叫以守卫领地和吸引雌鸟。无论是雌鸟还是雄鸟，它们的上半身都为绿色，尾部的羽毛为带金属光泽的蓝绿色。雌鸟的面颊和胸部羽毛为棕色，而雄鸟的下半身羽毛为鲜红色。雏鸟羽毛的颜色与雌鸟类似，不过翅膀尖部为独特的白色。

O’Reilly 图书封面上的很多动物都是濒危动物，它们对于这个世界很重要。如果想了解如何提供帮助，请访问网站 animals.oreilly.com。

本书封面图片来自 *Wood’s Animate Creation*。